A comparative study of Scrum and Kanban approaches on a real case study using simulation

David J. Anderson[1], Giulio Concas[2], Maria Ilaria Lunesu[2], Michele Marchesi[2], and Hongyu Zhang[3]

[1] David J. Anderson&Associates inc., Seattle, WA, USA
[2] DIEE - Department of Electrical and Electronic Engineering, University of Cagliari, Piazza D'Armi, 09123 Cagliari, Italy
[3] School of Software, Tsinghua University, Beijing, China

Abstract. We present the application of software process modeling and simulation using an agent-based approach to a real case study of software maintenance. The original process used PSP/TSP; it spent a large amount of time estimating in advance maintenance requests, and needed to be greatly improved. To this purpose, a Kanban system was successfully implemented, that demonstrated to be able to substantially improve the process without giving up PSP/TSP. We customized the simulator and, using input data with the same characteristics of the real ones, we were able to obtain results very similar to that of the processes of the case study, in particular of the original process. We also simulated, using the same input data, the possible application of the Scrum process to the same data, showing results comparable to the Kanban process.

Key words: Scrum, Kanban, Lean software development, software process simulation.

1 Introduction

A well-defined software process can help a software organization achieve good and consistent productivity, and is important for the organizations long-term success. However, an ill-defined process could overburden developers (e.g., with unnecessary meetings and report requests) and reduce productivity. It is thus very important to be able to understand if a software process is efficient and effective.

Many software processes have been adopted in industrial practices. For example, the Personal Software Process (PSP) [4] proposed by SEI shows software engineers how to plan, track and analyze their work. The Team Software Process (TSP) [5] is built on the PSP and extends it to a software project team. Scrum [13] is an iterative, incremental software process, which is by far the most popular Agile development process [15]. In recent years, the Lean-Kanban approach [1] advocates to minimize the Work-In-Process (WIP, which is the number of items that are worked on by the team at any given time) and to maximize the value produced by an organization.

Often the impact of a software process on software productivity is understood through actual practices. The evaluation of the effectiveness of agile practices, for instance, was performed by Maurer and Martel [6] and by Moser et al [9]. To be able to estimate the impact of processes before a project start, many software process simulation methods haven been proposed over the years. For example, Barghouti and Rosenblum [3] proposed methods for simulating and analyzing software maintenance process. Otero et al. [10] use simulation to optimize resource allocation and the training time required for engineers and other personnel. Melis et al. [8] [7] proposed an event-driven simulator for Extreme Programming practices such as test-driven programming and pair programming. In a previous work [2], some of the authors presented an event-driven simulator of the Kanban process and used it to study the dynamics of the process, and to optimize its parameters.

To help better understand and compare the software processes including the original PSP/TSP, Scrum, and Lean-Kanban processes, in this paper we propose a software process simulator, which can describe the details of a software project (e.g., features, activities, developers, etc.) and simulate how a process affect the project. Our simulator is an event-driven, agent-based simulator. We use it to simulate the PSP, the Scrum and the Lean-Kanban processes for software maintenance activities.

The simulations are performed over a Microsoft case study, which is based on four years of experiences of a Microsoft maintenance team in charge of developing minor upgrades and fixing production bugs. Our simulation results show that the Lean-Kanban approach can increase the efficiency of maintenance activities. These results are consistent with the actual experiences of the Microsoft team. After one year from the introduction of Lean-Kanban approach, this team was able to finish the outstanding work and to reduce the average time needed to complete a request. Our simulation work confirms that a WIP-limited approach such as Lean-Kanban can indeed improve maintenance throughput and improve work efficiency.

The paper is organized as follows: in Section 2, we introduce the case study and the related processes (PSP, Scrum, and Kanban). In Section 3, we describe our process simulator and its applications to the studied processes. We present the Microsoft case study and show the simulation results in Section 4. Section 5 concludes the paper.

2 The case-study and the related processes

This paper analyzes a real software development case study, where a transition was made from a traditional software engineering approach based on PSP to a WIP-limited Lean approach. We use real data from this case study to assess the software process simulator we developed, and as an input to a Scrum process simulation, to verify the possible results of the use of Scrum in the process.

The case study regards a maintenance team of Microsoft, based in India and in charge of developing minor upgrades and fixing production bugs for about 80

IT applications used by Microsoft staff throughout the world. It has already been described by one of the authors in the chapter 4 of [1], because it was one of the first applications of the WIP-limited approach described in that book, making use of a virtual kanban system. Note that there was no kanban board, because the board was not introduced until January 2007 in a different firm. The success of the new process in terms of reduced delivery time and customers' satisfaction has been one of the main factors that raised interest on such Kanban approach in software engineering.

The process is not about the development of a new software system, or about substantial extensions to existing systems, but it deals with maintenance, the last stage of the software life cycle. The importance of maintenance is well known, because it usually counts for the most part of the system's total cost – even more than 70% [16]. The typical maintenance process deals with a stream of requests that must be examined, estimated, accepted or rejected; the accepted requests are implemented updating the code, and then verified through tests to assess their effectiveness and the absence of unwanted side-effects.

In the following subsections we will briefly describe the original process used by the team, the new Kanban-based process, and a possible Scrum process applied to the same team.

2.1 The original process

The maintenance service subject of our case study is Microsoft's XIT Sustained Engineering, composed of eight people, including a Project Manager (PM) located in Seattle, and a local engineering manager with six engineers in India. The service was divided in two teams – development team and testing team, each composed of three members. The teams worked 12 months a year, with an average of 22 working days per month. The PM was actually a middle-man. The real business owners were in various Microsoft departments, and communicated with the PM through four product managers, who had responsibility for business cases, prioritization and budget control.

The maintenance requests arrived scattered in time, with a frequency of 20-25 per month. Each request passed through the following steps:

1. Initial estimate: this estimate was very accurate, and took about one day for one developer and one tester. The estimate had to sent back to its business-owner within 48 hours from its arrival.
2. Go-No go decision: the business owner had to decide whether to proceed with the request or not. About 12-13 requests per month remained to be processed, with an average effort of 11 man day of engineering.
3. Backlog: the accepted requests were put in a "backlog", a queue of prioritized requests, from which the developers extracted those they had to process. Once a month, the PM met with the product managers and other stakeholders to reprioritize the backlog.
4. Development phase (aka Coding): the development team worked on the request, making the needed changes to the system involved. This phase accounted on average for 65% of the total engineering effort. Developers used

TSP/PSP Software Engineering Institute processes, and were certified CMMI level 5.
5. Testing phase: the test team worked on the request to verify the changes made. This phase accounted on average for 35% of the total engineering effort. Most requests passed the verification. A small percentage was sent back to the development team for reworking. The test team had to work also on another kind of item to test, known as production text change (PTC), that required a formal test pass. PTCs tended to arrive in sporadic batches; they did not take a long time, but lowered the availability of testers.

Despite the qualification of the teams, this process did not work well. The throughput of completed requests was from 5 to 7 per month, averaging 6. This meant that the backlog was growing of about 6 request per month. When the team implemented the virtual kanban system in October 2004, the backlog had more than 80 requests, and was growing. Even worse, the typical lead times, from the arrival of a request to its completion, were of 125 to 155 days, a figure deemed not acceptable by stakeholders.

2.2 The Lean-Kanban process

To fix the performance problem of the team, a typical Lean approach was used. First, the process policies were made explicit by mapping the sequence of activities through a value stream, in order to find where value was wasted. The main sources of waste was identified in the estimation effort, that alone was consuming around 33 percent of the total capacity, and sometimes even as much as 40 percent. Another source of waste was the fact that these continuous interruptions to make estimates, which were of higher priority, hindered development due to a continuous switching of focus by developers and testers.

Starting from this analysis, a new process was devised, able to eliminate the waste. The first change was to limit the work-in-progress and pull work from an input queue as current work was completed. WIP in development was limited to 8 requests, as well as WIP in testing. These figures includes an input queue to development and testing, and the requests actually under work. Then, the request estimation was completely dropped. The business owners had in exchange the possibility to meet every week and chose the requests to replenish the input queue of requests to develop. They were also offered a "guaranteed" delivery time of 25 days from acceptance into the input queue to delivery.

In short, the new process was the following:

1. All incoming requests were put into an input backlog, with no estimation.
2. Every week the business-owners decided what request to put into the input queue of development, respecting the limits.
3. The number of requests under work in both activities – development and testing – were limited. In each activity, requests can be in the input queue, under actual work, or be finished, waiting to be pulled to the next activity.

4. Developers pulled the request to work on from their input queue, and were able to be very focused on a single request, or on very few. Finished requests were put in "Done" status.
5. Testers pulled the "Done" requests into their input queue, respecting the limits, and started working on them, again being focused on one request, or on very few. Requests that finished testing were immediately delivered.

This approach was able to substantially increase the teams' ability to perform work, substantially lowering the lead time from commitment and meeting the promised SLA response of 25 days or less for 98% of requests. Commitments were not made until a request was pulled from the backlog into the input queue.

Further improvements were obtained by observing that most of the work was spent in development, with testers having a lot of slack capacity. So, one tester was moved to the development team, and the limit of development activity was raised to 9. This further incremented the productivity. The team was able to eliminate the backlog and to reduce to 14 days the average cycle time.

2.3 The Scrum process

Scrum is by far the most popular Agile development methodology [15]. For this reason we decided to evaluate the introduction of Scrum for managing the maintenance process. A hypothetical introduction of Scrum would be similar to the Kanban approach, eliminating the estimation phase in exchange for a shortened cycle time. A typical Scrum process would be:

1. Incoming requests are put into a backlog. The Product Managers would act as the Product Owners, and the PM would act as the Scrum master (albeit) remote from the engineering team. The requests are prioritized by the Product Owners.
2. The development and testing proceeds through time-boxed iterations, called Sprints.
3. At the beginning of each Sprint, the Product Owners chose a given number of requests to implement in the Sprint. These requests are presented and estimated by developers and testers in a Sprint Planning Meeting.
4. Development and testing is performed on these requests during the Sprint, that is closed by a Sprint Review Meeting. The finished requests are delivered, while those still under work are passed to the next iteration.

Of course, we have no data about the adoption of Scrum for the maintenance process. However, we may make some observations about it. The first is that, even in the best case of a team able to self-organize giving more resources to coding with respect to testing, the cycle time cannot go below the iteration length. The meetings before and after each iteration would last at least one day, so it is better to have iteration lengths not too short – say at least two or three weeks – not to spend too much time in meetings. In general, we expect Scrum to produce relatively similar results – maybe just a little less effective. An important point is that Scrum was not a viable choice for *"political"* reasons,

because it was considered non-compatible with PSP or TSP, or both. Note that a recent work claims that Scrum and PSP can indeed be used together [12]. The Kanban system was not seen in this way, because PSP was not replaced but merely augmented with the Kanban system.

3 Agent-based Process Modeling

To model the software maintenance process, we used an approach that can be described as event-driven and agent-based. It is event-driven because the operation of the system is represented as a chronological sequence of events. Each event occurs at an instant in time and marks a change of state in the system [11]. It is also agent-based because it involves the representation or modeling of many individuals who have autonomous behaviors (i.e., actions are not scripted but agents respond to the simulated environment) [14]. In our case the agents are the developers, but in a broad sense also the activities can be considered as entities that can change their behavior depending on the environment. For instance, the activities will not "accept" requests in excess of their limits, that can vary with time. The basic entities of the proposed model, common to all simulated processes, are the requests, the activities and the team members.

The maintenance request are atomic units of work. They are characterized by an arrival time, expressed in days after the starting day of the simulation, an effort that represents the man days needed to implement and test the request, a priority in a given range, and a state, representing the completion state of the request within each activity. The requests can be taken from real records, or can be randomly generated. In this case study they are randomly generated, using statistic parameters taken from the real data. All the requests have the same priority, because requests were prioritized by deciding on which of them the work had to be started, and not by assigning explicit priority values.

The activities represent the work to be done on the requests. They are ordered and are characterized by a name, a limit on the number of requests that can be handled in the activity at any given time, and the typical percentage of the total estimated cost of a request that pertains to the activity. In our model the activities are, in the order:

- **Planning**: it represents the choice of the maintenance requests on which to start the work. This activity implies no engineering effort, and puts the chosen issues in the "Input Queue" to the subsequent activity.
- **Development**: it studies the work to be done to the existing system (bug fixing or enhancement), and perform this work by changing the source code of the target system and producing a new executable. This activity accounts for 65 percent of the total work on a request.
- **Testing**: the last activity, where the work made is verified and accepted. If the request does not pass the verification, it is be sent back to the Development phase for reworking. The percentage of rejected requests was very low, also due to the high qualification of the development team (CMMI level 5). In this

study we will not consider this case. This activity accounts for 35 percent of the total work on a request.

The team members represent the engineers working on the requests in the various activities. Each member has a name, an availability state (available, non-available), and a "skill"in each of the relevant activities. If the skill is equal to one, it means that the team member will perform work in that activity according the declared effort – for instance, if the effort is one man day, she will complete that effort in one man day. If the skill is lower than one, for instance 0.8, it means that a one-day effort will be completed in $1/0.8 = 1.25$ days. A skill lower than one can represent an actual impairment of the member, or the fact that she has also other duties, and is not able to work full-time on the requests. If the skill in an activity is zero, the member will not work in that activity.

In this case study the engineers can either be developers, with skill equal to one in Coding, and equal to zero in Testing, or testers, with skill equal to zero in Coding, and equal to 0.95 in Testing. This lower figure accounts for the time devoted by testers to test PTC requests (see the previous section), that are not considered as explicit requests in this study.

Each team member works on a single request (in a specific activity) until the end of each day, or until the work on the request in that activity is completed. When a new request is introduced, or work on a request ends, and in any case at the beginning of a new day, the system looks for idle engineers, and tries to assign them to requests available to be worked on in the activities they belong to. A request is handled by only one team member at any given time.

An important concept related to the work on requests is that of *penalty factor*, p. The penalty factor p is equal to one (no penalty) if the same team member, at the beginning of a day, works on the same request s/he worked the day before. If the member starts a new request, or changes request at the beginning of the day, it is assumed that s/he will have to devote extra time to understand how to work on the request. In this case, the value of p is greater than 1 (1.3 in our case study), and the actual time needed to perform the work is divided by p. For instance, if the effort needed to end work on a request in a given activity is t' (man days), and the skill of the member is s, the actual time, t, needed to end the work will be $t = \frac{t' s}{p}$. If the time extends over one day, it is truncated at the end of the day. If the day after the member will work on the same request of the day before, p will be set to one in the computation of the new residual time. The probability q that a member choses the same request of the day before depends on the number of available requests in the member's activity, n_r. In this case study we, set this probability equal to one for all considered processes.

A more detailed description of the specific events of the simulation, and of the general model can be found in [2], in the context of Kanban process simulation.

3.1 The model of the original process

To model the original process, we introduced at the beginning of each day (event "StartDay") a check of the new requests. If one or more new requests arrived

that day, one developer and one tester are randomly chosen, and their availability is set to "false" until the end of the day. In this way, we modeled the estimation work of accepted requests. We also modeled the estimation of not accepted requests by randomly blocking for a day a couple formed by a developer and a tester, with probability equal to the arrival rate of not accepted requests (about p = 0.45).

We set the maximum number of requests in the "Coding" phase at 50, not to flood this activity with too many requests.

3.2 The model of the Scrum process

To model the Scrum process, we had to introduce in the simulator the concept of iteration. To this purpose, we introduced the event "StartIteration", that takes place at the beginning of the day when the iteration starts. This event sets to "false" the availability of all developers and testers for a given time T_S, to model the time needed to hold the review meeting of the previous Sprint, and the Sprint planning meeting of the current one. T_S was set to one day in the considered case study.

Since the Scrum team is able to self-organize, and since the bottleneck of the work flow is coding, the Scrum team should self-organize to accommodate this situation. So, in the Scrum model we modeled all engineers both as developers and testers, in practice merging the two teams into one. In this way, coding is no longer the bottleneck, and the work is speeded up. This assumption gives a significant advantage to Scrum over other processes.

At the beginning of each Sprint, a set of request is taken from the Backlog and pulled into the Planning activity, to be further pulled to Coding. These request are chosen in such a way that the sum of their effort is equal to, or slightly lower than, a given amount of "Story points" to implement in each iteration. The requests that were still under work at the end of the previous Sprint are left inside their current activity, and their remaining effort is subtracted by the available Story points. The activities have no limit, being the flow of requests naturally limited by the Sprint planning.

4 Results and Discussion

We simulated the three presented models using data mimicking the maintenance requests made to the Microsoft team presented above. We generated two sets of requests, covering a time of four years each (1056 days, with 22 days per month). The average number of incoming requests is 22.5 per month, with 12.5 accepted for implementation, and 10 rejected. So, we have 600 accepted requests in total, randomly distributed. One of the sets had an initial backlog of 85 requests, as in the case study when the process was changed, while the other has no initial backlog.

The distribution of the efforts needed to complete the requests is Gaussian, with an average of 10 and a standard deviation of 2.5. In this way, most requests

have an estimated effort between 5 and 15. Note that the empirical data show an average effort per request of about 11 man days. In fact, at least in the original process, the engineers were continuously interrupted by estimation duties, with a consequent overhead due to the application of the "penalty" for learning, or relearning the organization of the code to modify or to test. In practice, we found that the average effort to complete a request was about 11 in the simulation of the original process. This value is equal to the empirical evaluation of 11 "engineering man days" needed on average to complete a request.

For each of the three studied processes, we performed a set of simulations, with the same data in input. For each process, and each input dataset, the outputs tends to be fairly stable, performing several runs with different seeds of the random number generator. For each simulation, we report the cumulative flow diagram (CFD), that is the cumulative number of requests entering the various steps of the process, from "Backlog" to "Released", and statistics about the cycle time. The cycle time starts when work begins on the request – in our case when it is pulled to the "Coding" activity, and ends when it is released.

In the followings we report the results for the three processes.

4.1 The original process

Fig 1 shows a typical CFD for the data of the original process. This diagram was obtained using the dataset with no initial backlog, and then rescaling the initial time to the time when the backlog of pending requests reached the value of 85, that is at day 287 from the beginning of the simulation.

The figure makes evident the inability of the process to keep the pace of incoming requests. The throughput of the team is about 6 request per month, and the backlog of pending requests grows of about 6.5 per month. These figures exactly correspond to the empirical value measured on real data. The "Coding" line represents the cumulative number of requests entered into the Coding activity, while the "Testing" line represents the cumulative number entered into the Testing activity. Having limited to 50 the maximum number of requests in the Coding allow to have a relatively limited WIP. The cumulative number of released requests (red line) is very close to the Testing line, meaning that the time needed to test the requests is very short. The slope of the red line represent the throughput of the system.

Table 1. Statistics of cycle times in the Original Process

Time Interval	Mean	Median	St.Dev.	Min	Max
200-250	140.72	131.49	76.2777	35.02	371.53
251-300	150.18	151.03	79.72	12.61	364.89
301-350	170.34	168.65	89.89	9.96	363.23
351-400	162.65	120.16	88.58	64.51	334.69

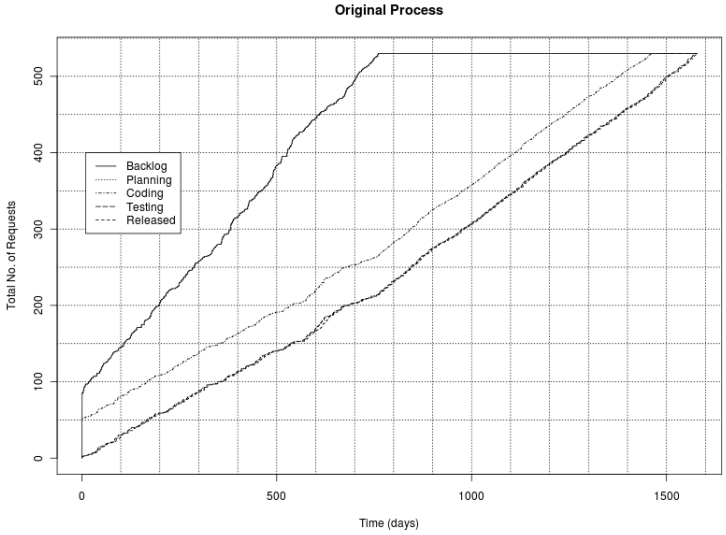

Fig. 1. ***The CFD of the original process.***

If we allow one tester to become also a developer, increasing the flexibility of the team, the throughput increases to 7.3 requests per month. Adding one developer and one tester to the teams, keeping the above flexibility, further increases the throughput to 8.1 requests per month, a figure still too low to keep the pace of incoming requests.

In Table 1 we report some statistics about cycle time in various time intervals of the simulation. In general, these statistics show very long and very variable cycle times. We remember that the backlog of pending requests reaches the value of 85, when the process was changed, at day 287. Around this time, the average and median cycle times are of the order of 150-160, values very similar to those reported for real data.

So, we can conclude that the simulator is able to reproduce very well the empirical data both in term of throughput and of average cycle time.

4.2 The Kanban process

In the case of Kanban process, the input dataset includes an initial backlog of 85 requests, with no work yet performed on them. The process was simulated by moving a tester to the developer team after 6 months from the beginning of the simulation (day 132), as it happened in the real case. The activity limits were set to 8 (9 from day 132) and 8 for Coding and Testing, respectively, as in the real case.

The resulting CFD is reported in Fig. 2. Note the slight increase in the steepness of the Coding and Testing lines after day 132, with a consequent in-

crease of the contribution made by Testing to the WIP. With the adoption of the Kanban system, the throughput substantially increases with respect to the original process. Before day 132 the throughput is about 10 requests per month (30 per quarter); after day 132 it increases to about 12 requests per month (36 per quarter), almost able to keep the pace of incoming requests.

If we compare the throughput data with those measured in the real case (45 per quarter in the case of 3 + 3 teams, and 56 per quarter in the case of 4 + 2 teams), we notice that in the real case the productivity is 50 percent higher than in the simulated process. Our model already accounts for the elimination of estimations, and for not having penalties applied the day after the estimation. Note that the maximum theoretical throughput of 6 developers working on requests whose average is 10 man days is 13.2 per month, and thus 39.6 per quarter, not considering the penalties applied when a request is tackled for the first time both during coding and testing. In the real case, there were clearly other factors at work that further boosted the productivity of the engineers. It is well known that researchers have found 10-fold differences in productivity and quality between different programmers with the same levels of experience). So, it is likely that the same engineers, faced with a process change that made them much more focused on their jobs and gave them a chance to put an end to their "bad name" inside Microsoft, redoubled their efforts and acheved a big productivity improvement.

Regarding cycle times, their statistics are shown in Table 2, for time intervals of 100 days starting from the beginning of the simulation. These times dropped with respect to the original situation, tending to average values of 25.

We also simulated the Kanban process with an increase of both team sizes of one unit, after 8 months from its introduction, as in the real case. We obtained an increase of throughput to 14.7 requests per month, or 44 per quarter, with the average cycle time dropping to 14.3.

Table 2. Statistics of cycletime in the Kanban Process

Time Interval	Mean	Median	St.Dev.	Min	Max
1-100	25.18	22.74	14.06	6.98	146.72
101-200	28.99	27.97	12.92	11.69	87.53
201-300	24.41	22.05	8.15	11.62	49.18
301-400	26.39	24.13	10.37	11.23	78.34

4.3 The Scrum process

We simulated the use of a Scrum process to manage the same input dataset of the Kanban case, that includes the initial backlog. In the presented case study, we choose iterations of 3 weeks (14 working days, accounting for the day spent in

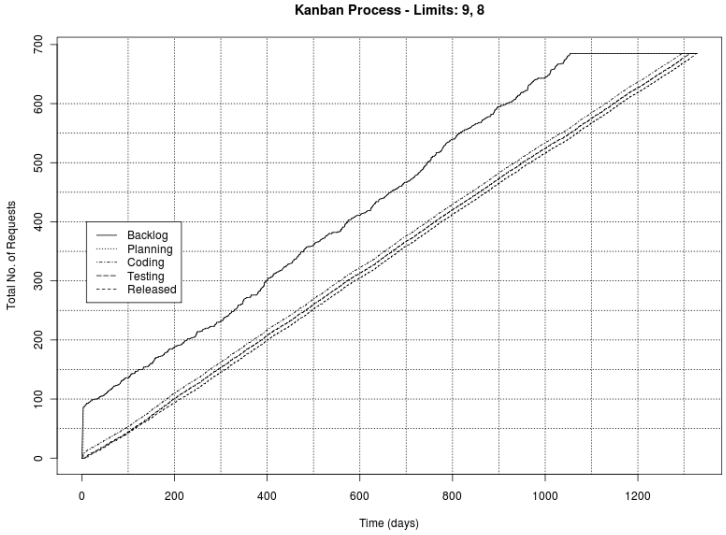

Fig. 2. ***The CFD of the Kanban process.***

meetings) because it is the minimum time span accommodating requests whose average working time is more than 10 man days, and with about 15% of the requests longer than the average plus a standard deviation, so more than 12.5 engineering days. Remember the constraint that only one developer works on a request at a time – a constraint mimicking the way of work of the actual teams. With a two-week iteration the team should spend a lot of time to estimate the length of the requests, and in many cases it should split them into two smaller pieces to do them sequentially across two Sprints. Even with a 3 week Sprint some requests would need to be split, but we do not explicitly handle this case – simply, the remaining of the request is automatically moved to the next iteration.

The number of Story points to implement is set to 90. In fact, with 14 working days per team member during a Sprint, we have a total of 84 man days. We slighly increased this limit to 90, to accomodate variations. We found empirically that a further increase of this limit does not increment throughput. We remember that, in our model of Scrum, all 6 engineers are able to perform both coding and testing (the latter with 0.95 efficiency), thus modeling the self-organization of the team.

Fig. 3 shows the CFD diagram in the case of Scrum simulation. Note the characteristic "ladder" appearance of the Coding line, that represents the requests in input to each Sprint. This process is much better than the original one, and is almost able to keep the pace of incoming requests, with a throughput of about 11.5 requests per month. This should be compared with the throughputs of Kanban with both teams of 3 engineers (10) and with 4 developers and 2 testers (12).

Had we not allowed the team to "self organize", the throughput would have been much lower.

The cycle time statistics are shown in Table 3. They are better than in the Kanban process, owing the highest team flexibility. Note that in our simulation, we do not wait for the end of the Sprint to release finished requests, but release them immediately. If we had waited until the end of the Sprint, as in a "true" Scrum implementation, all these average times should be increased of 3 days (50% of the difference between the Sprint length and the minimum cycle time, that is about 9). This is the average waiting time of a request between its completion and the end of the Sprint. Anyway, the Scrum results are very good, and comparable with the Kanban ones.

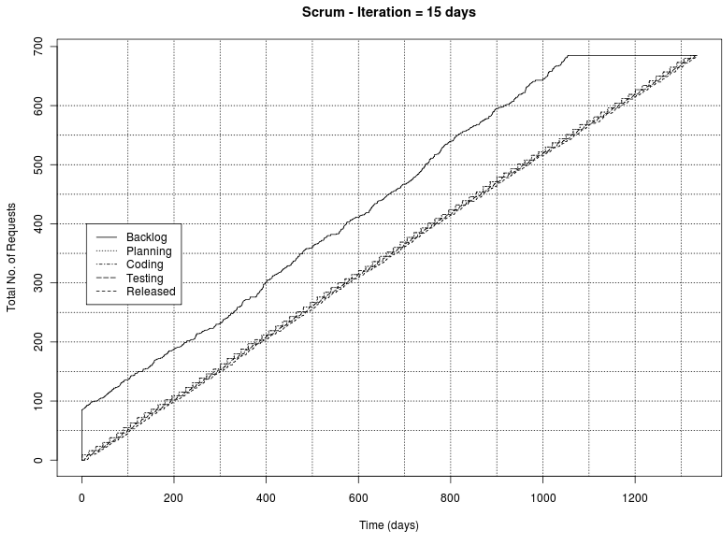

Fig. 3. ***The CFD of the Scrum process.***

Table 3. Statistics of cycle time in the Scrum Processes

Time Interval	Mean	Median	St.Dev.	Min	Max
1-100	16.69	15.74	4.65	8.62	28.00
101-200	16.68	15.79	5.90	9.03	34.51
201-300	16.41	15.71	4.72	9.72	30.50
301-400	16.79	16.41	4.92	8.91	28.02

5 Conclusions and future work

We presented the application of the process simulation model developed for assessing the effectiveness of agile and lean approaches described in [2] to a real case study, in which a maintenance team experimented the transition from a PSP/TSP, estimation-based approach to a Lean-Kanban process. We added also the modeling and simulation of a possible application of the Scrum process to the same case study, albeit Scrum was not really tried in the real case.

The proposed approach to model and simulate a software process, using an agent-based, fully object-oriented model, demonstrated very effective. It allowed us to model all three processes with minimal changes in the model and in the simulator. The use of a general-purpose OO language like Smalltalk eased this task, allowing a high flexibility in extending the simulator.

We used as input data a stream of maintenance requests synthetically generated, with the same statistical properties of real requests. We were able to fully reproduce the statistics of empirical results for the original process, both in terms of throughput and cycle times.

Regarding Kanban, the simulation results demonstrated a substantial improvement with respect to the original process, but in fact significantly lower than the improvement obtained in the real case, where the throughput was 50% higher than in the simulated results. This fact is worth further study, because in the real case it was reported a throughput that can be explained only with an increase in the productivity of the engineers of the team. Such an increase in individual productivity is a parameter difficult to introduce *a priori* in a simulation, without being accused of wishing to favour a process with respect to another.

The proposed simulation approach allowed us to easily model and apply to the case study also the Scrum process, despite its iterative nature, different from the "steady flow" nature of the two other processes. Since Scrum is based on self-organizing teams, we modeled this fact by eliminating the difference between developers and testers, a possibility suggested by the fact that in the real case a tester was actually moved to the development team. This self-organization made Scrum an advantage in flexibility reflected in the lower average and maximum cycle time with respet to Kanban. On the contrary, the simulated Kanban process – when the teams are better balanced, with 4 developers and 2 testers – still overcomes Scrum in throughput.

Overall, we believe that the presented work demonstrated that our agent-based approach is very effective for modeling and simulation of agile and lean software development processes, that tend to be simple and well structured, and that operate on a backlog of "atomic" requirements. This is particularly true for maintenance processes, that naturally operate on an inflow of independent requests.

In the future, we plan to further evaluate our simulation method on a variety of software development and maintenance projects, including open source projects, with the aim to explore the optimal parameter settings that can maximize the overall development efficiency. We will devote a specific effort to analyze

and model human factors that could affect the productivity of a development team, in relation with the specific process and practices used.

References

1. Anderson D.J.: Kanban: Successful Evolutionary Change for Your Technology Business, Blue Hole Press, 2010.
2. Anderson, D.J., Concas, G., Lunesu, M. I., and Marchesi, M.,.Studying Lean-Kanban Approach Using Software Process Simulation. Proc. Agile Processes in Software Engineering and Extreme Programming 12th International Conference, XP 2011, Madrid, Spain, May 10-13, 2011.
3. Barghouti, N. S., Rosenblum, D. S..: A Case Study in Modeling a Human-Intensive, Corporate Software Process. *Proc. 3rd Int. Conf. On the Software Process* (ICSP-3), 1994, IEEE CS Press.
4. Humphrey W. S., Using a defined and measured Personal Software Process, IEEE Software, May 1996, pages 77-88.
5. Humphrey W. S., Introduction to the Team Software Process, Addison Wesley, 1999.
6. Maurer, F., Martel, S.: On the productivity of agile software practices: An industrial case study. Technical report, Univ. of Calgary, Alberta, March 2002.
7. Melis M., Turnu I., Cau A. and Concas G.: Evaluating the Impact of Test-First Programming and Pair Programming through Software Process Simulation. Software Process Improvement and Practice, vol. 11, 2006, pp. 345-360.
8. Melis M., Turnu I., Cau A. and Concas G.: Modeling and simulation of open source development using an agile practice. Journal of Systems Architecture, vol. 52 , 2006, pp. 610-618.
9. Moser R., Abrahamsson P., Pedrycz W., Sillitti A., and Succi G.: A case study on the impact of refactoring on quality and productivity in an agile team. *Proc. IFIP Central and East European Conference on Software Engineering Techniques*, Poznan, Poland, 2007, Springer LNCS 5082.
10. Otero, L.D., Centeno, G., Ruiz-Torres, A.J., Otero, C.E..: A systematic approach for resource allocation in software projects. Comput. Ind. Eng. 56 (4) (2009) 1333-1339.
11. Robinson, S.: Simulation – The practice of model development and use. Wiley, Chichester, UK, 2004.
12. Rong G., Shao D., and Zhang H.: "SCRUM-PSP: Embracing Process Agility and Discipline", *Proc. 17th Asia-Pacific Conference on Software Engineering, APSEC 2010*, IEEE Press., pp. 316-325.
13. Schwaber, K., Beedle, M. Agile software development with Scrum. Prentice Hall (2002).
14. Siebers,P. O., Macal, C. M., Garnett,J., Buxton, D., and Pidd, M.,: Discrete-event simulation is dead, long live agent-based simulation!. Journal of Simulation (2010) 4, pp. 204-210.
15. Version One.: State of Agile Survey 2010. Online at www.versionone.com.
16. Wolverton, R.W.: The Cost of Developing Large-Scale Software. IEEE Trans. on Computers, vol 23, 1975, pp. 615-636.