Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2013; **25**:1225–1248
Published online 16 May 2013 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1599

Simulation of software maintenance process, with and without a work-in-process limit

Giulio Concas[1], Maria Ilaria Lunesu[1], Michele Marchesi[1,*,†] and Hongyu Zhang[2]

[1]*Department of Electrical and Electronic Engineering, University of Cagliari, Cagliari, Italy*
[2]*School of Software, Tsinghua University, Beijing, China*

SUMMARY

A software maintenance process is important for reducing maintenance effort and for improving software quality. In recent years, the Lean–Kanban approach has been widely applied in software practice including software maintenance. This approach minimizes Work-in-Progress (WIP), which is the number of items that are worked on by the team at any given time, thus improving the maintenance process. In this paper, we describe our simulation studies, which show that the Lean–Kanban approach can indeed help reduce the average time needed to complete maintenance requests. We develop a process simulator that can simulate both existing maintenance processes that do not use a WIP limit and that adopt it. We perform two case studies using real maintenance data collected from a Microsoft project and from a Chinese software firm. The main results of our study are twofold. First, we demonstrate that it is possible to effectively model and simulate, using actors and events, a maintenance process where a flow of issues is processed through a sequence of activities, correctly reproducing key statistics of real data. Second, our results confirm that the WIP-limited process could be useful to increase the efficiency of software maintenance, as reported in previous industrial practices. Copyright © 2013 John Wiley & Sons, Ltd.

Received 15 October 2012; Revised 16 January 2013; Accepted 30 March 2013

KEY WORDS: software maintenance process; software process simulation; Lean software development; Kanban; empirical software engineering

1. INTRODUCTION

Maintenance is an important stage of software life cycle that accounts for a large percentage of a system's total cost. After a software product is released, the maintenance phase keeps the software up to date with respect to discovered faults, environment changes and changing user requirements. A software maintenance team often has to cope with a long stream of requests that must be examined, answered and verified.

In practice, these activities are often performed with no specified process, or using heavyweight processes [1] that overburden developers with meetings, report requests, item prioritization and so on. Therefore, it is very important to adopt a maintenance process that is efficient and easy to follow.

Recently, the Lean software development (LSD) approach has been applied to software practice, including software maintenance. It is an Agile approach whose underlying concept is borrowed from manufacturing [2]. The Lean approach is focused on maximizing the value given to the customer and on reducing waste at every level. One of the key practices of Lean is to minimize Work-in-Progress (WIP), which is the items that are worked on simultaneously by the team at any given time. In this way, developers are maximally focused, and the waste of time and resources because of

*Correspondence to: Michele Marchesi, Department of Electrical and Electronic Engineering, University of Cagliari, Cagliari, Italy.
†E-mail: michele@diee.unica.it

switching from one item to the other is minimized. A tool to minimize WIP – and to increase the information flow about the project status among the team – is the Kanban board [3]. In general, we can define the Kanban software process as a WIP-limited pull system visualized by the Kanban board. Recently, the Kanban method is attracting a growing interest among software developers [4]. One of the first known applications of the Kanban method was conducted by a maintenance team in Microsoft India. After 1 year from the introduction of Kanban, this team was able to finish the outstanding work items and to reduce the average time needed to complete a maintenance request from 155 to 14 days [3].

Ideally, the best way to evaluate the efficiency of the Lean–Kanban approach would be to collect empirical data from a software organization before and after adopting this new process and then compare the data. Unfortunately, the Lean–Kanban approach is relatively new, and the number of Kanban applications is still limited, so actual data before and after adoption are scarce and are typically kept inside software firms as trade secret.

In order to help verify the efficiency of a WIP-limited software maintenance process as advocated by a Lean–Kanban approach, we propose a simulation-based method, which is the focus of this paper. We first develop a generic simulator and tune it to reflect the original maintenance process. We then limit the number of maintenance requests (i.e., WIP) a maintenance team can work on at each given time and use the tuned simulation to simulate a WIP-limited maintenance process. Our simulation study shows that the WIP-limited process can lead to an improvement in throughput. Furthermore, the WIP-limited process outperforms the original process that does not use a WIP limit.

In this paper, we present case studies on two maintenance projects. The first case study is on the project described in [3], which is based on 4 years of experience of a Microsoft maintenance team in charge of developing minor upgrades and fixing bugs. Our simulation work confirms that the WIP-limited process can indeed improve maintenance throughput and work efficiency. These results are consistent with the actual experiences of the Microsoft team.

The second project is from a Chinese software firm. We find that the original maintenance process is quite inefficient – such as the original Microsoft maintenance process cited above. We design a software process simulation model for post-release maintenance activities. Through simulations, we show that we can improve the original process by introducing a WIP limit. We assess the capability of the model to reproduce real data and find that the WIP-limited process could be useful to increase the efficiency of the maintenance process, as happened in the Microsoft project.

The main results of our study are twofold. First, we demonstrate that it is possible to effectively model and simulate a maintenance process, where a flow of independent issues is processed through a sequence of activities. Our simulation model, which is based on actors and events, is able to reproduce key statistics of real data, such as throughput, lead and cycle time. Second, our results confirm that the WIP-limited process as advocated by the Lean–Kanban approach could be useful to increase the efficiency of software maintenance, as reported in previous industrial practices.

The paper is organized as follows: In Section 2, we give an overview about existing modeling and simulation work on software maintenance; in Section 3, we describe the simulation model; Section 4 portrays the first case study on the Microsoft maintenance project, and Section 5 presents the second case study on the Chinese maintenance project. Section 6 describes the threats to validity. Section 7 discusses the implications of the work on process and project management. Section 8 concludes the paper.

2. BACKGROUND AND RELATED WORK

2.1. Lean–Kanban approach to software engineering

Recently, the Lean approach is becoming popular among software engineers. Its underlying concepts were first introduced in manufacturing in Japan between 1948 and 1975 [5, 6]. In 2003, Poppendiecks [2] published a book about LSD that applied Lean principles to software production. They identified seven key Lean principles: eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people and optimize the whole [7, 8].

Perhaps the most important principle of LSD is to eliminate waste. Waste is defined as anything that does not add value to a product, value as perceived by the customer [2]. In software development, there are several kinds of waste, for example: defects, unnecessary features, delays, partially completed work, task switching and extra processes.

In LSD, an important tool for managing workflows and controlling waste is the concept of pull system [2], where new work is pulled into the system or into an activity when there is capacity to process it, rather than being pushed into the system when a new request arrives. In this context, LSD strives to keep the WIP limited. This allows developers to focus on a few tasks, reducing the waste of time and resources because of switching from one item to another. The main Lean tool to visualize the flow and to keep WIP in control is the Kanban board.

A Kanban board graphically shows the activities of the process in its columns. The items under work are represented by cards attached on the board in the column representing the item's current state. Each activity (column) has a maximum number of items that can be pulled into it, thus controlling WIP. The board allows the project team to visualize the workflow, to limit WIP at each workflow stage and to monitor the cycle time (i.e., the time to complete working on an issue, from start to finish). A related tool is the cumulative flow diagram, which shows how many work items are processed in a specific time window for a specific phase of development. In this context, Petersen and Wohlin introduced and validated a set of metrics connected to cumulative flow diagrams [9]. These measures are aimed to control throughput and lead time (i.e., the time to complete resolving an issue, from its report to finish) and to provide a tracking system that shows the progress/status of software product development.

Ikonen *et al.* [10] empirically investigated the impact of Kanban adoption on a development team from various perspectives, suggesting that Kanban motivates the workers and helps managers to control the project activities. Wang, Conboy and Cawley studied the recent shift of attention from Agile to Lean [11]. Their analysis showed that Lean can be applied in agile processes in different manners for different purposes. Lean concepts, principles and practices are most often used for continuous agile process improvement. In some cases the Kanban approach has been used to substitute a continuous, flow-based process to time-boxed agile processes.

Sjberg, Johnsen and Solberg compared the usage of Scrum and Kanban in a medium-sized firm across a period of 2 years [12]. They found that by using Kanban instead of Scrum, the company almost halved the lead time, reduced the number of weighted bugs by 10%, improved productivity by 21% for implemented features and reduced bugs by 11%. Although these results should be interpreted with caution, they suggest that Lean–Kanban approach eliminates the waste of periodic Scrum meetings and the rigidity of Scrum time-boxing, while keeping all other advantages of the agile approach.

2.2. *Software maintenance process simulation*

Simulation is a generic term that includes a set of methods and applications representing how real systems work, typically using a general-purpose computer. The use of a simulator can help to recreate a real scenario, to analyze and evaluate the performances of different systems and to identify bottlenecks or critical points. Software process simulation is a research topic that was initially developed more than 30 years ago and is still active [13, 14].

The concept of simulation has been also applied to software maintenance. For example, Barghouti and Rosenblum [15] presented a case study on the feasibility, usefulness and limitations of a process support tool to model and analyze a real maintenance process. They especially studied the maintenance processes adopted by large software development organizations. Marvel was the environment they used for simulation, guidance, tracking and querying of the status of the maintenance process. Marvel was one of the first systems for modeling and simulating software processes. Antoniol *et al.* [16] proposed an approach based on queuing theory and stochastic simulation to deal with staffing and management of cooperative, distributed maintenance projects. They demonstrated their approach using data collected from a massive Y2K maintenance project on a large COBOL/JCL financial software system of a European firm.

In [17], search-based techniques for optimal resource allocation in massive maintenance projects were evaluated. A simulation approach based on complex dynamics and rule-based approaches was used to analyze transient and dynamic behavior of maintenance. Lin and Huang [18] applied queuing theory to model software fault correction activities through simulation with the aim to analyze staffing levels and costs.

2.3. Lean–Kanban process simulation

In the literature there are some simulation models of the Lean–Kanban approach for manufacturing processes. For example, Huang *et al*. [19] assessed the issues in adapting Japanese Just in Time techniques to American firms using network discrete simulation. Hurrion [20] performed process optimization using simulation and neural networks. Kochel and Nielnder [21] proposed a data-driven Kanban simulator called KaSimIR. Hao and Shen [22] proposed a hybrid simulator for a Kanban-based material handling system.

The simulation approach has been also used to simulate agile development processes. For example, event-driven simulators for Extreme Programming practices were introduced by Melis *et al*. [23, 24]. A Petri-net-based simulation of Lean–Kanban approach was cited by Corey Ladas on his website [25]. In a previous work, we presented an event-driven simulator of the Kanban process, from which we derived the simulator presented in this paper [26]. We used this simulator on synthetically generated data to show the advantages of a WIP-limited approach versus a non-limited one and to optimize the parameters of the process, namely the WIP limits of the various activities of the process. In a subsequent work we used an extended version of the simulator to compare Lean–Kanban with traditional and Scrum approaches on the data collected from a Microsoft project, showing that the Lean–Kanban approach is superior to the others [27]. This paper extends our previous work for modeling and evaluating the Lean–Kanban-based software maintenance process.

Recently, Turner *et al*. worked on modeling and simulation of Kanban processes in Systems Engineering – the engineering of complex or evolving systems composed of hardware, software and human factors [28, 29]. These work, though quite preliminary, propose the use of a mixed approach, merging Discrete Event and Agent-based simulation approaches. In particular, Discrete Event simulation is used to simulate the flow of high-level tasks and the accumulation of value, whereas Agent-based simulation is used to model workflow at a lower level, including working teams, Kanban boards, work items and activities.

3. SIMULATING THE SOFTWARE MAINTENANCE PROCESS

To verify the efficiency of a WIP-limited software maintenance process, as advocated by a Lean–Kanban approach, we make use of simulation models. Generally, the typical activities of a maintenance process are as follows:

1. Planning: It represents the choice of the maintenance issues (including bug-fixing requests and enhancement requests), on which to start the work. This activity typically takes a short time and puts the chosen issues in the Input Queue to the subsequent activities.
2. Development: It represents the development work to be done to the existing system (including bug fixing and enhancement). This activity can be further divided into the Analysis and Coding phases.
3. Verification (testing): It represents the work for verifying changes made to address the issues.

During software maintenance, a stream of issues is generated. The issues arriving at any given time are firstly put in a backlog queue of the system. The issues are then processed through the sequence of activities cited above, each consuming a given percentage of the total effort needed to complete the issue. In Lean–Kanban approach, each activity is represented by a column in the Kanban board, holding the cards representing the issues under work in the activity. The column is in turn divided in two vertical areas, from left to right – the issues under work in the activity, and the issues completed (Done) and waiting to be pulled to the next activity.

Figure 1 shows an overview of a software maintenance process. A maintenance project is aimed to resolve a set of issues, and these issues are processed in a sequence of activities (with or without limits on the maximum number of issues in each activity). The maintenance work is performed by a team of developers. Each developer is able to work in one or more activities, but only on one issue at a time. Issues may not pass the test phase, and thus, could be sent back to a previous activity to be reworked.

3.1. Generic simulation model for software maintenance processes

The work items in a maintenance project are bug-fixing or enhancement requests, called issues. They correspond to the features described in the Lean–Kanban approach. Each issue is characterized by a unique identifier, a report date, an effort expressing the actual amount of work needed to complete the issue in man days, and a priority, which is a number in a given range, expressing importance of the issue; a higher number corresponds to a higher priority.

The modeled maintenance process has the following characteristics:

1. The simulation starts at time zero. Time is expressed in days. Each day involves 8 hours of work. At the beginning, the system may already hold an initial backlog of issues to be worked out.
2. Issues are entered at given times, drawn from a random distribution or given as input to the system.
3. Each issue is assigned an effort to be fixed (in days of work). This value can be drawn from a distribution, or obtained from real data.
4. Each issue passes sequentially through the phases of Planning, Development (Analysis and Coding) and Verification, as described above. It is possible that a given percentage of issues pass directly from an activity to the final closed state (we observed this behavior in one of the real-world maintenance processes empirically studied in this paper). Each phase takes a percentage of the whole effort to process the issue. The sum of the percentages of the three phases is 100%. When an issue enters an activity, the actual effort (in man days) needed to complete the activity is equal to the total effort of the issue multiplied by the percentage.
5. The number of team developers may vary with time, with developers entering and leaving the team.
6. The developers working on the issues in the activity may have different skills. 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, the member will complete that effort in one man day. If the skill is lower than one, for instance 0.8, it means that 1-day effort will be completed in $1/0.8 = 1.25$ days. A skill lower than one can represent an actual impairment of members, or the fact that they have also other duties, and are not able to work full time on the issues. If the skill for an activity is zero, the member will not work in that activity.

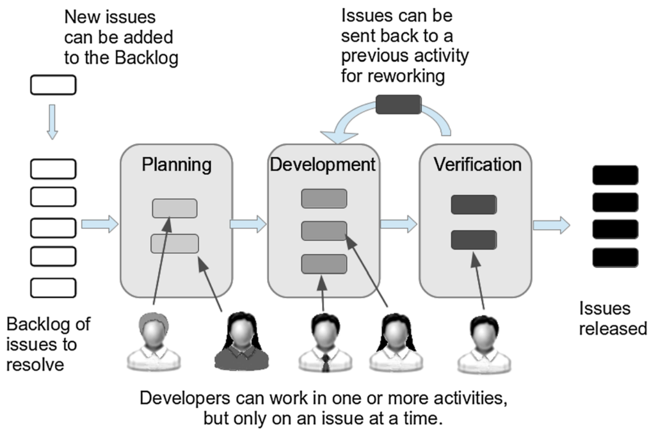
New issues can be added to the Backlog
Issues can be sent back to a previous activity for reworking
Planning
Development
Verification
Backlog of issues to resolve
Issues released
Developers can work in one or more activities, but only on an issue at a time.

Figure 1. An overview of a maintenance process that consists of a sequence of activities that may have limits on the number of issues under work.

7. At the beginning of the day, each developer picks an issue in one of the activities s/he can work on. The issues are picked at random, taking into account their priority. This is obtained by sorting issues by their priority plus Gaussian noise (with zero mean and standard deviation *s*) that accounts for variability in priority management. The lowest priority issue is assigned with a probability of being picked up proportional to 1, whereas the highest priority one is assigned with a probability proportional to $pm \leq 1$. All intermediate issues are assigned a probability proportional to a value between 1 and *pm*, with a linear interpolation on their priority. In this way, the issues with the highest priority will be typically processed first, but leaving a chance also for the lower priority ones. This process continues until there are no issues in that activity, or to the end of the 8-h working day.
8. When an issue is processed for the first time, or when the work on the issue is resumed by a developer who is different from the one who worked on it on the previous day, a penalty $p > 1$ is applied to the work, to model the waste because of task switching (extra time needed to study the issue, which is proportional to the size of the task). In our model, the time actually needed to solve the issue is the remaining effort multiplied by the penalty. When the effort to complete an issue in a given activity is low – a few hours – the penalty is applied when work is started on the issue for the first time, but probably the work will anyway end within the day. When the effort is bigger, the work in an activity will take more than one day. If the developer working on the issue is the same across the days, the penalty is applied only for the first day. If not, it can be repeatedly applied, and the overall work will be longer. All developers who worked on an issue in the previous day try to continue their work on it with a given probability that we call *pp*, divided by the number of issues ready to be worked on in the activities pertaining to the developer. Clearly, when there are many issues to work on in an activity, the chance for a developer to work on the same issue of the previous day is smaller than when these issues are few.
9. When the work on an issue in a given activity ends, the issue is passed to the next activity. When Verification ends, the issue becomes closed.

This is a generic simulation model for representing many maintenance processes that can be customized to cater for a specific maintenance process of an organization. From the generic model, we can derive two specific models, one with a WIP limit as suggested by the Lean–Kanban approach, and the other one without a WIP limit as adopted by current common practices. For a WIP-limited process, the model has to be complemented by adding limits to the maximum number of issues than can be processed at any given time inside an activity.

3.2. The design of the simulator

We designed a simulator to simulate the software maintenance process. Our simulator is event-driven, meaning that the simulation proceeds by executing events, ordered by their time. When an event is executed, the current time of the simulation is advanced to the time of the event. The simulator holds an event queue, where events to be executed are sorted by their time (day, hour, minute, second). When an event is executed, it can change the state of the system, create new events (with times equal to, or greater than, the current time) and insert the new events into the event queue. The simulation ends when the event queue is empty, or if a given date and time is reached.

The time used in our simulation is recorded in nominal working days, from the start of the simulation. It can be fractionary, denoting days partially spent. The simulation does not explicitly account for weekends, holidays, or overtime. A day is considered to have eight nominal working hours. The relevant events of the simulation are the followings:

- Creation: a new issue is created and inserted into the backlog, and a ToPull event is generated for the first activity, at the same time. This event refers only to issues introduced after the start of the simulation, and not to the issues initially included in the backlog.
- WorkEnded: the work of a developer on an issue, within a given activity, has ended and the developer becomes idle. This may happen when the work on the issues is actually completed regarding the activity, or at the end of the working day. In the former case, the state of the issues is changed, and a ToPull event is generated for the next activity, at the same time.

- ToPull: an activity is requested to pull an issue from the previous activity, or from the backlog, if it is the first. If the activity is nil, it means that a completed issue should be moved from the last activity to the Closed state. If the activity has already reached its maximum number of issues, or if there are no issues to pull in the previous activity, nothing happens. If an issue can be pulled, it is pulled to the activity, and another ToPull event is generated for the previous activity (if the activity is not the first), at the same time. All the idle developers of the team are asked to find issues ready to be worked on. If a developer finds such an issue, the developer is assigned to the issue, the actual time taken to complete the work is computed, and a WorkEnded event is generated and inserted into the event queue for the time when the work will end (at the end of the current day, or when the work on the issue is completed).

The three events are enough to manage the whole simulation. The simulation starts by creating the starting issues, putting them in the backlog, generating as many ToPull events (at $time = 0$) for the first activity as its maxIssues value, and then generating as many Creation events for future times as required. The simulator is then asked to run, using the event queue created in that way. When the work on all issues has been completed in all activities, no more issues can be pulled and no more work can start, the event queue becomes empty and the simulation ends.

The model was implemented using the Smalltalk object-oriented language. A typical simulation of the whole maintenance process (5000 requests and 1100 days) takes about 13 s on a Dual-Core Intel 2.35 GHz PC with 3.4 GB of RAM, running Linux.

4. CASE STUDY 1: A MICROSOFT MAINTENANCE PROJECT

To evaluate if a Lean–Kanban-based software maintenance process (which imposes a WIP-limit) can achieve a better performance than a conventional maintenance process that has no WIP limit, we performed two case studies. The first case study is on a maintenance project in Microsoft India. This project is about developing minor upgrades and fixing production defects for about 80 IT applications used by Microsoft staff throughout the world. As described in [3], it was one of the first applications of the WIP-limited software maintenance process. The success of the new process in terms of reduced delivery time and customer satisfaction has been one of the main factors that raised interest on the Lean–Kanban approach in software engineering.

4.1. Modeling the original process

The subject of our first case study is Microsoft's XIT Sustained Engineering, which was 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 into 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 over time, with a frequency of 20–25 per month. Each request passed through the following steps:

1. Planning: Once a request arrived, it was estimated by one developer and one tester. The estimate was sent back to its business-owner within 48 h from its arrival. 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 days. 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.
2. Development phase (including analysis and coding): the development team worked on the request, making the needed changes to the system involved. This phase accounted for 65% of the total engineering effort. Developers adopted Team Software Process (TSP) and Personal

Software Process (PSP) proposed by the Software Engineering Institute [30], and were certified CMMI level 5.

3. Verification/Testing phase: the test team worked on the request to verify the changes made. This phase accounted 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), which does not require 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 five to seven per month, with an average of six. This meant that the backlog was growing up about six requests per month. At the time the team started to implement the virtual Kanban system in October 2004, the backlog had more than 80 requests and was still growing. Even worse, the typical lead times, from the arrival of a request to its completion, were about 125 to 155 days, a number deemed not acceptable by stakeholders.

To model the original process, we introduced at the beginning of each day a check of the new requests, creating a new event StartDay. If one or more new requests arrive on 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 time spent to estimate 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 Development phase to 50, in order not to flood this activity with too many requests.

4.2. *Modeling the WIP-limited process*

To fix the performance problem of the team, a Lean–Kanban approach was adopted by the Microsoft team in October 2004. First, the process policies were made explicit by mapping the sequence of activities through a value stream, in order to find out where value was wasted. The main sources of waste were identified in the estimation effort, which alone was consuming around 33% of the total capacity, and sometimes even as much as 40%. Another source of waste was the fact that these continuous interruptions to make estimates, which were of higher priority, hindered development because of a continuous switching of focus by developers and testers.

Starting from this analysis, a new process was devised to eliminate the waste. The first change was to limit the WIP 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 include an input queue to development and testing, and the requests actually under work. Then, the estimation request was completely dropped. The business owners had in exchange the possibility to meet every week and choose the requests to add to the queue. 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, without estimation.
2. Every week the business-owners decided which requests to be put into the input queue of development, respecting the limits.
3. The number of requests under work in both activities – development and testing – was limited. In each activity, requests could be in the input queue, under actual work, or 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 focus on a single request, or on a few requests. Finished requests were set to Done status.
5. Testers pulled the Done requests into their input queue, respecting the limits, and started working on them, again they were able to focus on one request, or on a few requests. Requests that finished testing were immediately delivered.

This approach was demonstrated to be able to substantially increase the teams' ability to perform work, lower the lead time and meet the promised response (25 days or less, starting from the day the request was pulled from the backlog into the input queue) for 98% of requests. Further improvements were obtained by observing that most of the work was spent on development, whereas testers were not heavily loaded and had a lot of slack capacity. Consequently, one tester was moved to the development team and the limit of development activity was raised to nine. This further increased the productivity. The team was able to eliminate the backlog and reduce the average cycle time to 14 days.

In this case study we simulate the Lean–Kanban approach. In our simulations, the engineers can either be developers (with skill equal to one in Development, and equal to zero in Verification), or testers (with skill equal to zero in Development, and equal to 0.95 in Verification).

An important concept related to the work on requests is the penalty factor, p (as described in Section 3). 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 a 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 finish the 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

$$t = \frac{t' s}{p} \tag{1}$$

If the required time is over one day, it is truncated at the end of the day. If on the next day 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 chooses 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 computed this probability in the following way:

$$q = \begin{cases} 1 & \text{if } n \le 20, \\ \dfrac{20}{n} & \text{if } n > 20. \end{cases} \tag{3}$$

Note that by selecting those parameters, the engineers will always work on the same request of the day before, if the number of available requests is smaller than or equal to 20. This is a common practice in Kanban.

4.3. *Experiments and results*

We simulated the two models (with and without a WIP-Limit) using data mimicking the maintenance requests made to the Microsoft team. We generated two sets of requests, covering a time of 4 years each (1056 days, with 22 days per month). The average number of incoming requests is 12.5 per month (600 total). One set had an initial backlog of 85 requests (the number of requests in the backlog when the process was changed in October 2004), whereas the other had no requests in the backlog (the initial size of the backlog).

The effort needed to complete each request is drawn from a Gaussian distribution. The average effort value should match the empirical value of 11 man days. However, the simulator applies an overhead of 30% (i.e., the penalty factor p of Equation 1 is set to 1.3) when a request is worked on for the first time, and when the person in charge of the request changes (which often happens during the simulations). In fact, the engineers are continuously interrupted by estimation duties – as in the real case – so they work on the same request for two or more consecutive days only sporadically. For this reason, we had to set the average request effort value to a value smaller than 11, because the original effort is actually increased when the penalty is applied. We used an average effort value of 9.4 man days and a standard deviation of 2.5. In this way, 95% of the requests have an original (non-penalized) effort between 4.4 and 14.4 man days, and the actual average effort when penalties are applied turns out to be just about 11 man days.

For each of the two processes (with and without a WIP-Limit), we performed a set of simulations, using the same data as input. For each process and each input dataset, the outputs tend to be fairly stable when several runs with different seeds of the random number generator are performed. In the following subsections we report the results of the two processes.

4.4. Simulation of the original process

Figure 2 shows a typical WIP diagram for the data of the original process. The figure shows the inability of the process to keep pace with incoming requests. The throughput of the team is about six requests per month, and the backlog of pending requests grows of about 6.5 requests per month. These numbers exactly match the values of the real data. The Coding line represents the cumulative number of requests entered into the Coding activity, whereas the Testing line represents the cumulative number entered into the Testing activity. We limited the maximum number of requests in the Coding activity to 50 because the team never worked on more than 50 requests at the same time. The cumulative number of released requests (dashed line) is very close to the Testing line, meaning that the time needed to test the requests is very short. The slope of the dashed line represents the throughput of the system.

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, the throughput further increases to 10.1 requests per month, which is a figure still too low to keep pace with incoming requests.

In Table I we report some statistics about cycle time in various time intervals of the simulation. In general, these statistics show very long and variable cycle times. Note that the backlog of pending requests reached the value of 85, when the process was changed after 287 days. Around that time, the average and median cycle times are of the order of 150–160 days, very close to those reported for real data.

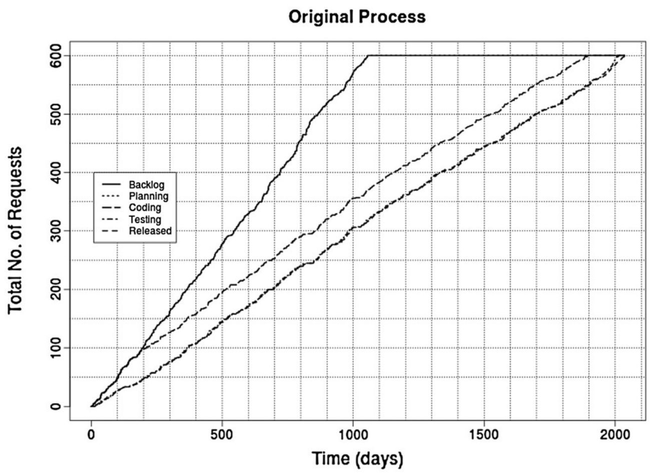

Figure 2. The WIP diagram of the original process.

Table I. Statistics of cycle times in the original process.

Time interval	Mean	Median	St. dev.	Min	Max
200–250	140.72	131.49	76.27	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

We can thus conclude that the simulator is able to reproduce very well the empirical data in terms of throughput and of average cycle time.

4.5. Simulation of the WIP-limited process

The input dataset of our WIP-limited process 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 11 and 8 for Coding and Testing, respectively, as in the real case.

The resulting WIP diagram is reported in Figure 3. Note the slight increase in the steepness of the Coding and Testing lines after day 132, with a consequent increase of the contribution made by Testing to the WIP. With the adoption of the Lean–Kanban approach, the throughput substantially increases compared with 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 enough to keep pace with 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. Moreover, because of the WIP limitation, developers tend to work on the same request in subsequent days, until it is completed, and thus the 30% penalty applied on the effort to learn the request is applied only for the first day. Note that the maximum theoretical throughput of 6 developers working on requests whose average effort to complete is 11 man days, is 12 per month and thus, 36 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 to an astonishing 56 requests per quarter. It is well known that researchers have found differences in productivity that may vary even of one order of magnitude among programmers with the same levels of experience, depending on their motivation and on the work environment (see for instance [31]). 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 achieved a big productivity improvement.

Statistics on cycle times are shown in Table II. The data are based on 100-day time intervals, starting from the beginning of the simulation. These times dropped compared with the original situation, leading to average values of 25.

We also simulated the WIP-limited process when adding one developer and one tester after 8 months from its introduction, as in the real case. We obtained an increase of throughput to 17.7 requests per month, or 53 per quarter, with the average cycle time dropping to 14 days. Figure 4

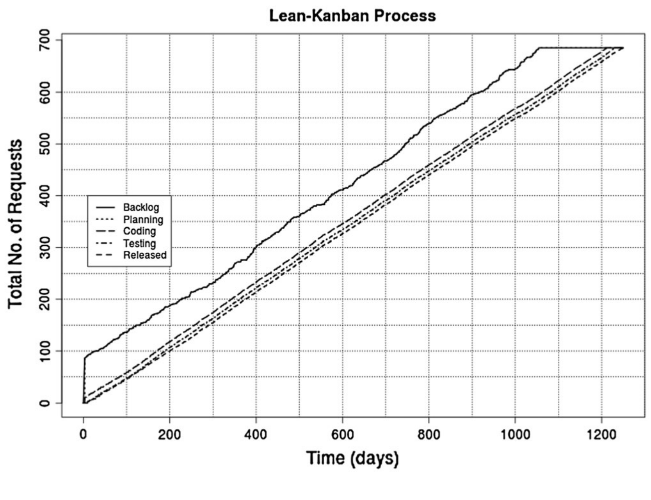

Figure 3. The WIP diagram of the WIP-limited process.

Table II. Statistics of cycle times in the WIP-limited 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

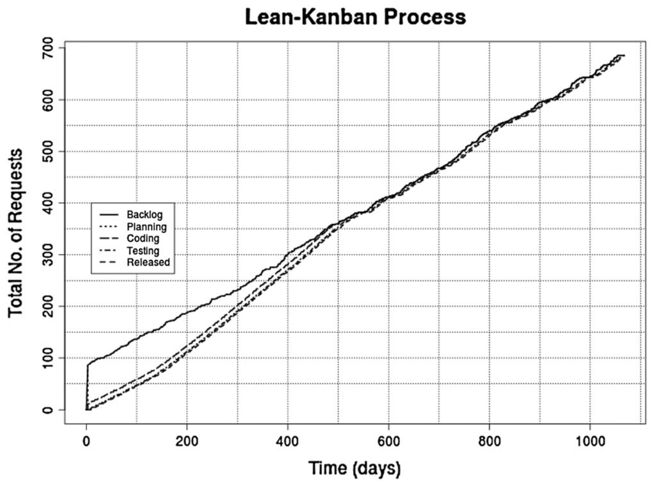

Figure 4. The WIP diagram of the WIP-limited process with a developer and a tester added after 6 months.

shows the cumulative flow diagram in this case, showing that the backlog is reduced to zero in 300 working days, which is in about 14 months.

We can conclude that the proposed simulation approach could effectively model the original process. As regards the WIP-limited process model, its efficiency clearly overcomes that of the original process, but not as much as in the real case. As already pointed out, most probably there were human factors at play, not directly related with the process, which further increased the developers' motivation and productivity.

5. CASE STUDY 2: THE MAINTENANCE PROCESS IN A CHINESE FIRM

To further evaluate if a WIP-limited software maintenance process can achieve better performance than a conventional maintenance process without using a WIP limit, we studied a large dataset describing maintenance activities at a Chinese IT company. The data cover the years between 2005 and 2010.[‡]

5.1. Data collection and analysis

In this company, when an issue is reported, an issue record is created. Each issue record includes a unique Id, a Report Date, an initial state Submitted, a priority and other essential information. In general, issues with higher priority are handled first; issues with lower priority values are dealt with later. Priorities are in the range from 0 to 30, with 30 being the maximum priority. A priority of zero means unspecified and will be dealt with as if it was the average of all possible priorities (priority equal to 15.5).

Upon arrival, the maintenance team analyzes the issue. They can judge that the issue is not worth further action, or is a duplicate of another reported issue. The issue might also be put on hold, waiting for further information (Suspended state). In most cases, analysis is followed by a coding

[‡]The dataset is available in a comma-separated text file at http://agile.diee.unica.it/data/MaintenanceData.csv

activity aiming to resolve the issue. When the maintenance team claims the issue is resolved, an Answer date is added to the issue record. Then, the modified software is passed to a verification team.

The verification team verifies the resolution of the issue. Sometimes, this team sends back the issue to the maintenance team because the verification was not successful. In most cases, the verification team closes the issue, setting the Verify Date. Whenever a change is made to the issue record, the Date of Change is set accordingly.

The dataset consists of 5854 records, each referring to an issue. For each record, there are 12 fields (not all set). The most relevant fields are shown in Table III.

There are 3839 CLOSED issues and 2015 OPEN issues. The Issue State of CLOSED issues can be only: ‘Resolved’, ‘NoFurtherAction’ and ‘Duplicate’. All issues have always set their Report Date and Date of Change; CLOSED issues have always set their Answer Date and Verify Date, whereas OPEN issues have often not set their Answer Date and Verify Date.

5.2. *Data analysis*

We performed a detailed analysis of the issue records. We limited our analysis to the period from 12/10/2007 to 30/9/2010 (the report date of the last issue) because there were a significant number of issues reported during this period. In Figure 5, we show the total number of issues entered into the system, exited from it and their difference (issues that were still under processing) from 12/10/2007 to 30/9/2010.

We can see from Figure 5 that the team devoted to resolve issues was not able to keep up with the pace of issue arrival for about 2 years (more than 700 days), until the number of issues waiting to be solved (Work in Progress, or WIP) reached the number of about 2000. Then, further resources were added to the team, so that it managed to keep up with new arrivals and this number did not grow further. This is also partially due to a slowdown of issue arrivals after day 800, which changed from about six issues/day to about three issues/day.

To obtain a deeper understanding of the maintenance efforts, we performed some more statistical analysis of the data as follows. First, we studied the time (in days) needed to work on an issue. The time information includes the total time from issue reporting date to verification (for CLOSED issues, also known as the lead time), the time from reporting to the answer date (that is the time the maintenance team needed to work on the issue) and the time from answer to verification (that is the time the verification team needed to work on the issue). Note that the collected time information includes not only the time needed to perform actual work but also the waiting times that are prevailing.

Table III. The main fields of the dataset studied.

Name	Description	Values
Issue ID	Unique identifier of the issue or bug to be fixed	Integer (6–7 digits)
Open/CLOSED	Information whether the issue has been closed or not	OPEN, CLOSED
Issue state	The current state of the issue	Resolved, Duplicate, NoFurtherAction, Submitted, Analyzed, NotAccepted, Suspended, Updated, Postponed, NeedMoreInfo
Report date	Date the issue was submitted to the maintenance team	Date (year–month–day)
Priority	Relative priority of the issue	Integer between 0 and 30, in increasing order. 0 means: unspecified.
Answer date	Date the team set an estimate for fixing the bug, after analysis	Date (year–month–day)
Date of change	Date of the last change to the record	Date (year–month–day)
Verify date	Date the Issue resolution has been verified by a verification team	Date (year–month–day)

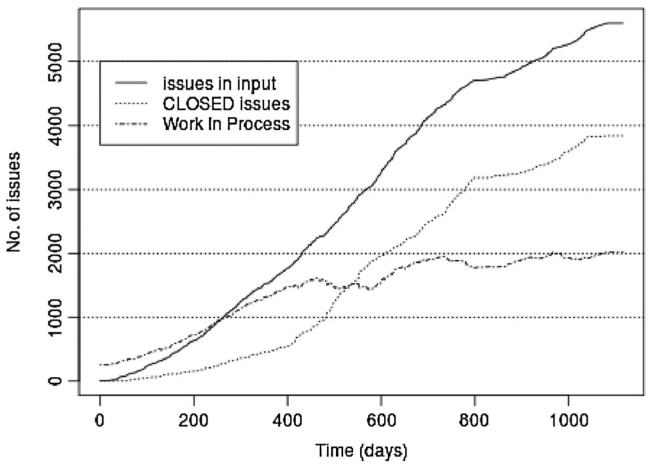

Figure 5. Cumulative number of issues in the system.

We found that the time needed to manage the issues exhibits large variations and is often longer than 1 year. This is compatible with the fact that the maintenance team was unable to cope with the arrival pace of reports for a long time and was never able to fill this gap. Table IV shows the main statistics of the considered times for closing, answering and verifying issues.

5.2.1. Analysis of developers and working time. We analyzed the total time required to manage issues. We were also interested in the actual working time spent on them. Unfortunately, we have no exact data about the actual number of developers belonging to maintenance and verification teams during the development. The information we were able to gather is as follows:

- In the beginning of the examined period, both teams were quite small (two to three developers, plus a few part-time testers);
- As the number of unsolved issues became critical, at about day 400, more developers were added (up to a number of seven to eight developers in the maintenance team);
- The maintenance team experimented a high turnover; in the end, it was composed of a few developers.

To assess in a quantitative way the number of developers, we used the empirical data. The first observation came from the firm that originated the data: the verification time is very small compared with the analysis and coding time, so we focused on estimating the size and working time of the maintenance team. We could observe that when the maintenance team cannot keep up with new issues, the size of the work in progress increases and the number of issues resolved depends only on the team's abilities, and not on the issues in input. From Figure 5, it is apparent that the number of issues completed follows the three different patterns as follows:

1. It is quite low until about day 400 (about 1.5 issues completed per day).
2. It suddenly increases between days 400 and 800 (to about 6.5 issues per day).
3. It slows down again to about 2.5 issues per day after day 800.

We make the hypothesis that the sudden variations in completion rates of issues for periods 1–3 were not because of substantial changes in issue quality, or in developers' skills, because we have no evidence of this. Consequently, we assume that such changes were due to a change in team size.

Table IV. Main statistics of time required to manage issues (in days).

Value	Median	Mean	St. dev.	Min	Max
Lead time	64	135.7	188.1	0	1729
Answer time	24	90.3	171.3	0	1694
Verification time	20	45.4	68.4	0	627

To compute the average actual working time to solve an issue, we have to account for the fact that the situation is not in steady-state, but closed issues are not able to follow the input flow of new issues, as highlighted in Figure 5. We considered that, for each closed issue, the team had also to work on other issues. We estimated at 50% the percentage of this extra-work with respect with the number of closed issues, which is roughly the percentage of issue still in progress at the end of the simulation (2015) compared with the closed issues (3839). Table V summarizes the hypotheses on team size and on the average actual work needed to complete issues in the examined periods.

5.3. *Experiments and results*

Using the proposed simulation method described in Section 3 and the data described in Section 5.1, we performed simulations for the original (without WIP-limits) and Lean–Kanban (with WIP-limit) processes.

The simulation models were evaluated for two purposes. The first purpose is verifying if it can generate similar output data as the original when its input data are the same as the original. The data we are referring to here are ‘arrival time’ and ‘priority’ as input data, and ‘number of solved issues as a function of time’ and ‘statistical properties of date’ as output data. The second is to explore, through the simulation, if the adoption of a WIP-limited approach can improve the maintenance process with respect to the overall number of resolved issues and the lead time for each resolved issue.

5.3.1. Simulation of the existing process. We first adapted the generic simulation model for software maintenance process as described in Section 3, so as to simulate the process reflected by the data presented in Section 5.1. The adapted process has the following characteristics:

1. Issues are entered at the same time as the empirical data, with the same priorities. Time zero is 12/10/2007. At time zero, the system has already 100 issue reports, taken from the latest 100 issue reports prior to 12/10/2007.
2. The effort of each issue is drawn from a distribution mimicking the distributions shown in Table V. The distributions represent the total number of days to close an issue, but we believe that the actual work time follows similar distributions. For the sake of simplicity we used a corrected lognormal distribution. The average of the original distribution is 1.1 and its standard deviation is 2.5. A correction is then made to these effort values, raising to 0.5 (half day of work) all efforts less than 0.5, because we deemed unrealistic to deal with issues needing less than 4 h of work to be fixed (including Analysis, Coding and Verification). The average of the corrected distribution thus becomes 1.255 and its standard deviation becomes 2.20. This is consistent with the estimates of the average time of actual work needed to fix a defect, reported in the last column of Table V.
3. The maintenance phases have the following specific characteristics:
   – Planning: Effort in this phase is considered negligible because issues are immediately passed to the Analysis phase as they arrive. After Planning, 1.5% of issues are immediately marked as CLOSED, reflecting the empirical percentage of issues with a lead time of 0 or 1 day.
   – Analysis: This phase is estimated to take 30% of the total effort. After Analysis, 5.4% of issues are immediately marked as CLOSED, reflecting the empirical percentage of issues with a cycle time of zero or one day after the Answer Date.
   – Coding: This phase is estimated to take 50% of the total effort.
   – Verification: This phase is estimated to take 20% of the total effort. In this study, for the sake of simplicity we do not consider issues that do not pass the Verification and are sent back to the Analysis phase.

Table V. Statistics about issue arrival, fixed issues flow and actual development work estimates.

Period (days)	New issues/day	Closed issues/day	Avg. team size	Issues/day + 52.5%	Avg. workdays/issue
1 (1–400)	6	1.5	2	2.3	1.15
2 (401–800)	6	6.5	8	9.9	1.24
3 (800–1049)	3	2.5	3	3.8	1.27

The percentages of 30–50–20 for Analysis, Coding and Verification efforts over total effort derive from the fact that we gave 20% of the overall effort to Verification. This is a very conservative assumption (in the sense that it is overestimated), because the estimates made by people working in the firm where the empirical data come from are typically lower, telling even that the verification of many bugs requires just 15 minutes. The 30–50 subdivision between analysis and coding was made following common software engineering knowledge [32]. However, changing these percentages does not have a substantial impact on the results, provided that the Verification quota does not increase.

4. The developers are divided into two teams: Maintenance Team (MT) and Verification Team (VT). The VT is devoted only to verification. The MT is composed of developers performing both Analysis and Coding – with no specific preference – but not Verification. This reflects actual work organization inside the studied firm. In practice, this is obtained using two kinds of developers: MT developers have skill equal to one in Analysis and Coding, and equal to zero in Verification. Vice-versa, VT developers have zero skill in Analysis and Coding, and one in Verification. The number of developers varies over time, reflecting the capacity to fix issues that varies in different periods of time. As explained in Section 5.1, and specifically in Table V, we considered three time intervals, whose length is 400, 400 and 284 days, respectively. They cover all the considered period of 1084 days. Table V reports, among other information, the number of developers devoted to maintenance and verification during the various phases. The factor *pm* (introduced in Section 3.1, for computing the probability an issue with the highest priority is picked at the beginning of the day) is set to 5. The standard deviation of the Gaussian noise added to priority is $s = 8$. In this way, the issues with the highest priority will be typically processed first, but leaving a chance also for the lower priority ones.
5. The penalty *p* is set to 1.5, meaning that the effort to fix the defect is increased by 50% to account for the time to understand the issue, to study the code and to fix it. The factor *pp* (for computing the probability a developer will continue the work on the same issue of the day before) is set to 200. This means that when available issues are less than or equal to 200, the developer will always choose to continue working on the same issue of the day before. If issues are more than 200, this choice is not granted. For instance, if there are 400 issues pending, there is only a 50% chance that a developer will choose the same issue of the day before. The value of 200 is probability overestimated, because even with a few issues to be chosen from, it is unlikely that the developer will stick on the same issue of the day before. However, we preferred to overestimate this value, not to give the Lean–Kanban approach (where developers choose among a limited set of issues) a too large advantage compared with the non-limited approach.

This model was implemented and simulated. Figure 6 shows the work flow diagram, which is the cumulative number of defects in the various possible states. Note that the first two states in this figure (Backlog and Planning) are coincident because there is no delay between reporting and planning. The Analysis, Coding and Verification curves show the cumulative number of issues that underwent the corresponding phase, or that are still in it. The Coding curve is well on the right of the Analysis one. Their horizontal distance shows the typical delay between the start of the Analysis phase and its ending, which is simultaneous to the start of the Coding phase. The Verification curve is on the far right of the plot. The horizontal distance between Coding and Verification shows the time spent in Coding. The CLOSED curve is higher than the Verification one, because it accounts also for the 6.9% of cases when a defect is closed upon its arrival, or just after the Analysis (as reported in the description of the characteristics of Planning and Analysis phases above). Without this artifact, it would be on the right of the Verification curve. Verification takes a short time because it accounts only for 20% of the whole processing time and because the test team does not lack developers.

Overall, in Figure 6, what really matters is the dotted curve representing CLOSED defects, which shows a good match to the dotted curve in Figure 5. The two solid curves in Figures 5 and 6 represent the cumulative number of reported defects and are, therefore, the same.

In conclusion, we believe that the simulated model produces data that match fairly well with the empirical ones, demonstrating the goodness of the approach in modeling and simulating real data.

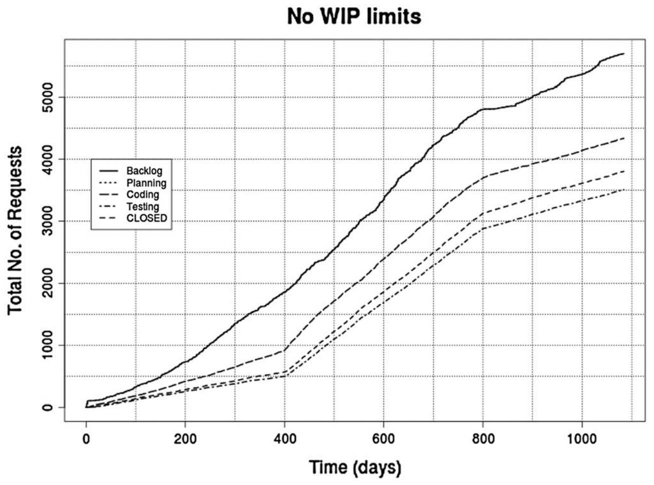

Figure 6. The WIP diagram without WIP limits. The Planning curve overlaps with the Backlog curve.

5.3.2. Simulation of the WIP-limited process. The WIP-limited model is built on the previous simulated model, but it enforces limits in the number of issues that can be worked on simultaneously in the various activities (in our model: Planning, Analysis, Coding and Verification). The characteristics of the WIP-limited model are as follows:

1. Issue requests are the same as those in the non-limited case.
2. Issue efforts and the probability that an issue is closed after Planning or Analysis are the same as those in the non-limited case.
3. The work related to each issue passes sequentially through the phases of Planning, Analysis, Coding and Verification, with the same relative percentages of efforts assigned to them as in the non-limited case. In each activity, an issue can be under work or done. Done issues wait to be pulled to the next activity, when there is capacity available. A maximum number of issues that each activity can bear (the WIP limit) are assigned to each activity. We tried many possible limits, finding that if they are high the results are very similar to the previous simulation (which is by definition unlimited). If limits are too small, work is hindered and slowed down. The best compromise is when no developer has to wait, but limits are minimized.
4. The developers are divided in two teams, of the same sizes and characteristics of the non-limited case, as reported in Table VI.
5. Selecting issues to be worked on follows the same style as it in the non-limited case, with the same policy for picking the issues at random, accounting for their priority.
6. The parameters *p* and *pp* are the same as those in the non-limited case. In this case, however, the probability to continue working on the same issue is $pp = 200$, divided by the number of issues contained in the activities pertaining to the developer; so in practice, it is always one because this number is limited and certainly, smaller than 200.
7. When the work on an issue in a given activity ends, the defect state is marked as done. It is passed to the next activity only when there is enough room available there (in order not to violate the

Table VI. Limits verified and actually used for the various activities during the simulation.

		Planning		Analysis		Coding		Verification	
Period (days)	Team sizes	Interval	Actual value	Interval	Actual value	Interval	Actual value	Interval	Actual value
1–400	(2, 1)	30–100	100	3–15	3	3–15	6	3–15	6
401–800	(8, 3)	50–150	100	8–25	10	8–25	10	6–20	8
801–1084	(3, 2)	40–100	100	3–15	10	3–15	10	4–15	6

WIP limits). When the choice is among many issues, the issue to pull is the one with the highest priority. When Verification ends, the issue immediately becomes CLOSED and is taken out from the maintenance process.

This model was implemented and extensively simulated, trying to assess the best values of the limits. As in the previous case, the Planning activity effort is considered negligible, so the issues entering Planning are immediately marked as done. However, these issues have to comply with the limits of this activity. New issues are pulled into Planning only when the number of issues belonging to it falls below one third of its limit – that is a sensible threshold for replenishing the buffer. This is tested at the beginning of each day.

The limits in the various activities obviously vary with the team sizes. They are set at the beginning of the simulation, then after 400 days and again after 800 days. With four activities, the total number of limits to set at the various times is 12, as shown in Table VI.

We performed hundreds of runs, varying most of the limits. The main goal was to increase the total number of defects closed after given periods of time. Note that the optimizations can be made step by step – first we can optimize the limits in the first 400 days, then the limits between 400 and 800 days, and eventually those in the last part of the simulation. Some results can be outlined:

- When limits are neither too small nor too high, it seems that they do not influence much the final number of closed issues.
- Limits in the first activity – Planning – are not important, provided they are high enough; they were set to reasonable values, in our case, of the order of one hundred.
- The last activity – Verification – is typically staffed with more developers than actually needed, at least in the present study; thus, its limits are much less critical than those of Analysis and Coding.

We report some typical values of the limits that seem to be the best in terms of the number of closed defects at the end of the simulation. Table VI shows the typical intervals we found to be best for each activity and each period. It also shows the actual values used in the simulations.

Figure 7 shows the cumulative number of defects in the various possible states for the WIP-limited simulation. This figure differs from Figure 6, being the four working states (from Planning to Verification) very close to each other and far from the Backlog (just reported) state. Also here the CLOSED curve is higher than the Verification one, because it accounts for the 6.9% of cases when a defect is closed upon its arrival, or just after the Analysis.

The Planning curve has a stepped appearance, because issues are pulled to this state only from time to time and in batches. The other three curves are almost overlapping, denoting a very short lead time to work on defects. Overall, the total number of defects closed at day 1084 is 4179, about 300 more than in the non-limited case. This is a good result, showing a higher productivity for the WIP-limited model. The main advantage of the WIP-limited approach is the increased system throughput.

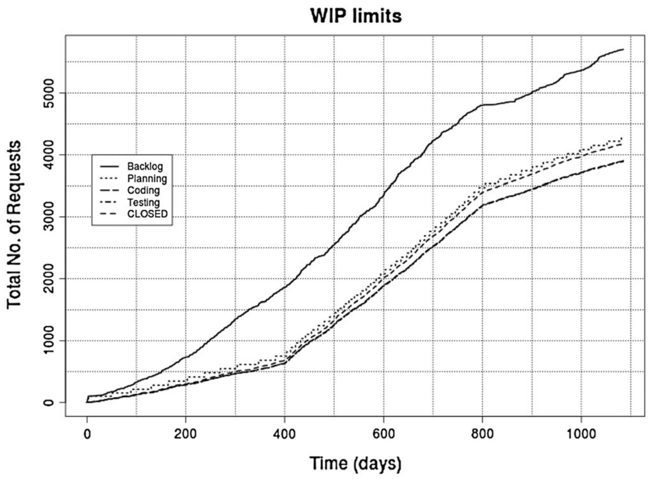

Figure 7. The WIP diagram with WIP limits. The Testing curve overlaps with the Released curve.

To better compare the ability of the two approaches, we executed 10 runs of the simulation for both the non-limited and the WIP-limited cases. In Figure 8, we show the number of issues in the CLOSED state, the average of 10 runs and their two standard deviation limits. We can see that the WIP-limited process is more efficient in closing the issues. At the end of the simulations, the average number of closed issues is 4145 in the WIP-limited case and 3853 issues in the non-limited case – that is about 7% less.

The higher efficiency of the WIP-limited approach is due to the fact that developers are more focused on fixing a few issues, because the number of issues they can work on is limited. In this way, it is less likely that they would change the issue at work the day after, and consequently, there is less overhead because of the penalty that is applied when the work on an issue is resumed by a different developer.

6. THREATS TO VALIDITY

In this paper, we present a method to model and simulate software maintenance processes, and its application on two case studies. In this section, we discuss what we consider to be the most important threats to validity that should be taken into consideration when applying the proposed method. There are three main types of threats to validity, namely construct validity, internal validity and external validity.

Internal validity: An experiment is said to possess internal validity if it properly demonstrates a causal relation between two variables, usually, the treatment and the outcome of the experiment [33]. In other words, there may be unknown, hidden factors that may affect the results. In our case, we have scarce information on the teams – the organizations that originated the maintenance data did not record detailed information on maintenance and verification teams. We were not able to obtain reliable data about composition, skills and percentage of time actually devoted to work on issues, and we had to extrapolate these data from the scarce available information. This might influence the quality of the results. Moreover, in the proposed simulation model, the issues are made through a sequence of activities performed by a team of developers, skilled in one or more of these activities. Therefore, our model fits well with the actual software maintenance process. Furthermore, our model does not consider the interactions among developers, which may have an impact on the maintenance efforts.

Construct validity: Construct validity concerns the degree to which inferences are warranted from the observed phenomena to the constructs that these instances might represent. The question,

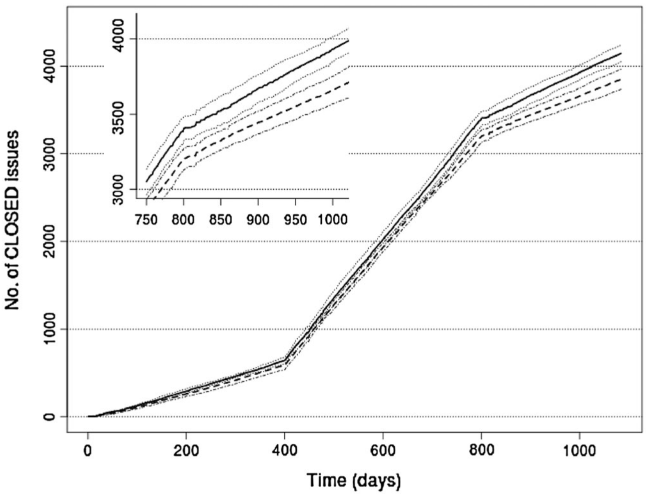

Figure 8. Comparison of the number of CLOSED issues versus time. The results are average of 10 runs, in WIP-limited case (solid line) and non-limited case (dashed line). The other lines represent two standard deviation intervals.

therefore, is whether the sampling particulars of a study can be defended as measures of general constructs [33]. A first threat to construct validity is that, although in this study we have carefully analyzed and preprocessed the Microsoft and Chinese maintenance data, our results could be affected by the data quality (such as possible noisy data). Another threat related to construct validity is the fact that our work is centered on the study of how the process determines the efficiency of the maintenance activity. However, there are many other human-related factors that could affect the efficiency and productivity of the team, such as, for instance, respecting the workers, keeping them motivated and satisfied, giving them all the tools they need and so on. Just limiting WIP will not be effective if the team is troubled and dissatisfied. A simulation model can simulate processes, but it is very difficult to explicitly include human factors.

External validity: An experiment is said to possess external validity if the experimental results hold across different experimental settings, procedures and participants. If a study possesses external validity, its results will generalize to a larger population not considered in the experiment [33]. In this study, we only ran the simulation model on two industrial maintenance projects. Although these projects are large, the number of subjects used in this study is small. This is because of the difficulty in collecting real industrial data. This is a clear threat to the external validity of our results. In future we will seek more data and perform more evaluations. Moreover, the simulation methods we proposed are evaluated on large software systems that have been experiencing a long period of evolution. For a small or short-living system, the number of maintenance requests is often small, thus making the simulation and statistical analysis unsuited.

7. DISCUSSION

In this paper we showed that: (i) when software maintenance process can be decomposed in resolving a set of relatively independent small issues and (ii) when issues can be implemented through a sequence of activities performed by a team of developers, modeling and simulating the maintenance process is possible. Using a model on the basis of software agents and simulating the process as a sequence of events enabled us to effectively reproduce the behavior of real software maintenance projects, even in presence of a change in the process, as shown in the Microsoft case study.

This fact has interesting implications and potential applications, at both process and project management levels. In this paper we mainly focus on the impact on software maintenance management. However, with proper modifications, we believe that most of our results could be also applied to a feature-oriented software development approach.

7.1. Process management implications

The effectiveness of a software maintenance process can be evaluated in terms of productivity (the throughput of resolved issues, weighted by their effort, per unit of time), average and maximum lead and cycle times (measuring how long it takes to give value to customers) and quality of produced software (number of defects and ease of maintenance). The proposed modeling and simulation approach is able to assess the effectiveness of a software process in terms of productivity and time needed to resolve the issues. It is well known that in software engineering, performing comparisons among processes is very difficult: empirical experiments using professional developers on real projects are very costly, and still cannot be completely controlled. The proposed approach enables these comparisons in a shorter time and at lower costs than approaches on the basis of empirical experiments. It thus could be a powerful tool for managing software maintenance policies in organizations. The proposed method could be also used to assess the impact of making incremental changes to an existing process, such as adding or removing prescribed meetings, or adding a more thorough quality assurance phase to the maintenance process.

When applied to general software development, the proposed method could be used to compare different development approaches, such as a traditional, waterfall-based approach, with an agile approach such as Scrum or Lean–Kanban. Some of these processes are already represented in the proposed model, other would require further modeling, but the main roles, activities, artifacts and events are already available, which are common to any software development process. In medium or large organizations, process management might also use the proposed simulation approach to assign developers to teams, and teams to projects, simulating the various options and choosing the best ones – those able to increase throughput, shorten the time to resolve issues and deliver value to the customers. It could help decide which process is better for new projects, considering the work items to be performed, the characteristics of the available team(s) and other constraints.

Overall, if an effective process that models a simulation tool was available, it would be very useful for managers to steer software maintenance and development policies.

7.2. Project management

At single project or single team level, the ability to simulate software maintenance effort can be a very useful tool for project managers. It would enable the estimation of cost and duration of maintenance requests. The estimate could be made up-front, at the beginning of a project, and could be tuned when the project is ongoing. Here tuning means both changing the parameters of the process to fit better existing data, and simulating the process from the current time on, using only new data (issues not completed, different team composition and so on). Such a tool might complement the existing estimation practices, which are mainly based on the experience of past projects and on the evaluation of requirements through methods such as Function Points [34] and its variants [35].

Another possible use of our approach at project level is the evaluation of the impact of changes while the project is ongoing. In fact, it is not uncommon that during the execution of a project a substantial change happens, such as large variations in requirements, changes in the team because of resignations and new recruitments, or the need that the team works on a different project at the same time. Being able to model and simulate the new situation would help the project manager assess the impact of the changes on project completion time and other key performance indicators.

8. CONCLUSION

Software maintenance typically accounts for a large percentage of software cost. It is important to adopt a maintenance process that is efficient and easy to follow. Recently, the Lean–Kanban approach has been applied in software practice including software maintenance. Industrial practices have shown that the Lean–Kanban approach can reduce maintenance and development costs by limiting the WIP items [3, 11, 12].

In this paper we presented a simulation model for software maintenance process. By simulation, we showed that a WIP-limited approach such as Lean–Kanban can indeed improve maintenance throughput and reduce cost. We performed two case studies on a Microsoft maintenance project and a Chinese maintenance project. These projects have gone through many years of maintenance. In each study, we first tuned the simulation model to simulate the existing, non-WIP-limited approach to maintenance, showing a good match between real and simulated data. We then simulated a WIP-limited approach to maintenance on the same data. The results show that a WIP- limited approach can improve maintenance efficiency.

In the future, we will further evaluate our simulation method on a variety of maintenance and development projects including open source projects, exploring the optimal settings that can maximize the overall maintenance efficiency.

We will also analyze and model the human and team interaction factors that could affect a project team's maintenance performance. This might be performed again using a simulation approach, by modeling these factors with a casual loop diagram and using System Dynamics [14]. Such an extension of the method could help model large productivity variations that are encountered in practice, which are difficult to model in an unbiased way.

Another substantial improvement to our model that we are considering is to scale the model from a single team to multiple teams involved in one project or even in several projects. This would greatly improve the utility of the tool for large organizations.

We are also actively working on the application of the model to risk management. This could not only evaluate the risk of a single project, but also mitigate the risk of different processes – such as Scrum and Lean–Kanban.

ACKNOWLEDGEMENTS

This research is supported by Regione Autonoma della Sardegna (RAS), Regional Law No. 7-2007, project CRP-17938 'LEAN 2.0', and the NSFC grant 61073006. We thank David Anderson for providing us the Microsoft project information and H.H. Wang for providing us the maintenance data from the Chinese firm.

REFERENCES

1. Jones C. The Economics of Software Maintenance in the Twenty First Century. (Available from: www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf) [14 February 2006].
2. Poppendieck M, Poppendieck T. Lean Software Development: An Agile Toolkit. Addison Wesley: Boston, 2003.
3. Anderson DJ. Kanban: Successful Evolutionary Change for Your Technology Business. Blue Hole Press: Sequim, WA, 2010.
4. Ladas C. Scrumban. Modus Cooperandi Press: Seattle, 2008.
5. Ohno T, Mito S, Schmelzeis J. Just-in-Time for Today and Tomorrow. Productivity Press: Cambridge, MA, 1988.
6. Womack JP, Jones D, Roos D. The Machine That Changed the World: The Story of Lean Production. Harper Perennial: New York, 1990.
7. Poppendieck M, Poppendieck T. Implementing Lean Software Development from Concept to Cash. Addison Wesley: Boston, 2006.
8. Poppendieck M, Cusumano MA. Lean Software Development: A Tutorial. *IEEE Software* 2012; **29**:26–32. DOI: 10.1109/MS.2012.107
9. Petersen K, Wohlin C. Measuring the flow in Lean software development. *Software Practice and Experience* 2011; **41**:975–996. DOI: 10.1002/spe.975
10. Ikonen M, Pirinen E, Fagerholm F, Kettunen P, Abrahamsson P. On the impact of Kanban on software project work – an empirical case study investigation. In 16th IEEE International Conference on Engineering of Complex Computer Systems. IEEE Computer Society Press: Las Vegas, NE, USA, 2011; 305–314. DOI: 10.1109/ICECCS.2011.37
11. Wang X, Conboy K, Cawley O. Leagile software development: an experience report analysis of the application of lean approaches in agile software development. *Journal of Systems and Software* 2012; **85**:1287–1299. DOI: http://dx.doi.org/10.1016/j.jss.2012.01.061
12. Sjberg DIK, Johnsen A, Solberg J. Quantifying the effect of using Kanban versus Scrum: a case study. *IEEE Software* 2012; **29**:47–53. DOI: 10.1109/MS.2012.110
13. Abdel-Hamid T, Madnick S. Software Project Dynamics: An Integrated Approach. Prentice-Hall: Upper Saddle River, NJ, 1991.
14. Madachy RJ. Software Process Dynamics. Wiley-IEEE Press: Chichester, UK, 2008.
15. Barghouti NS, Rosenblum DS. A case study in modeling a human-intensive, corporate software process. Proc. 3rd Int. Conf. On the Software Process (ICSP-3). IEEE CS Press: Reston, Virginia, USA, October 10–11, 1994; 99–110. DOI: 10.1109/SPCON.1994.344418
16. Antoniol G, Di Penta M, Harman M. Search-based techniques applied to optimization of project planning for a massive maintenance project. In 21st IEEE International Conference on Software Maintenance. IEEE Computer Society Press: Los Alamitos, California, USA, 2005; 240–249. DOI: 10.1109/ICSM.2005.79
17. Antoniol G, Di Penta M, Cimitile A, Di Lucca GA, Di Penta M. Assessing staffing needs for a software maintenance project through queuing simulation. *IEEE Transactions on Software Engineering* 2004; **30**(1):43–58. DOI: 10.1109/TSE.2004.1265735
18. Lin CT, Huang CY. Staffing Level and Cost Analyses for Software Debugging Activities Through Rate-Based Simulation Approaches. TR, Dec. 2009; 711–724. DOI: 10.1109/TR.2009.2019669
19. Huang PY, Rees LP, Taylor BW. A simulation analysis of the Japanese just-in-time technique (with Kanbans) for a multiline, multistage production system. *Decision Sciences* 1983; **14**:326–344. DOI: 10.1111/j.1540-5915.1983.tb00189.x
20. Hurrion RD. An example of simulation optimisation using a neural network metamodel: finding the optimum number of Kanbans in a manufacturing system. *Journal of the Operational Research Society* 1997; **48**:1105–1112.
21. Kochel P, Nielnder U. Kanban optimization by simulation and evolution. *Production Planning & Control* 2002; **13**:725–734. DOI: http://dx.doi.org/10.1016/j.ijpe.2004.06.046
22. Hao Q, Shen W. Implementing a hybrid simulation model for a Kanban-based material handling system. *Robotics and Computer-Integrated Manufacturing* 2008; **24**:635–646.
23. Melis M, Turnu I, Cau A, Concas G. Evaluating the impact of test—first programming and pair programming through software process simulation. *Software Process Improvement and Practice* 2006; **11**:345–360. DOI: 10.1002/spip.286
24. Melis M, Turnu I, Cau A, Concas G. Modeling and simulation of open source development using an agile practice. *Journal of Systems Architecture* 2006; **52**:610–618. DOI: http://dx.doi.org/10.1016/j.sysarc.2006.06.005

25. Ladas C. Kanban simulation. (Available from: http://leansoftwareengineering.com/2008/11/20/kanban-simulation) [31 December 2012].
26. Anderson DJ, Concas G, Lunesu MI, Marchesi M. Studying Lean–Kanban approach using software process simulation. Agile Processes in Software Engineering and Extreme Programming 12th International Conference, XP 2011. Springer LNBIP: Madrid, Spain, vol. 77, May 10-13, 2011; 12–26. DOI: 10.1007/978-3-642-20677-12
27. Anderson DJ, Concas G, Lunesu MI, Marchesi M, Zhang H. A comparative study of Scrum and Kanban approaches on a real case study using simulation. Agile Processes in Software Engineering and Extreme Programming 13th International Conference, XP 2012. Springer LNBIP: Malmoe, Sweden, vol. 111, May 21-25, 2012; 123–137. DOI: 10.1007/978-3-642-30350-09
28. Turner R, Ingold D, Lane JA, Madachy R, Anderson D. Effectiveness of Kanban approaches in systems engineering within rapid response environments. Conference on Systems Engineering Research (CSER). Procedia Computer Science: St. Louis, MO, vol. 8, March 19-22, 2012; 309–314. DOI: http://dx.doi.org/10.1016/j.procs.2012.01.065
29. Turner R, Madachy R, Ingold D, Lane JA. Modeling Kanban processes in systems engineering. 2012 International Conference on Software and System Process (ICSSP), June 2-3, 2012; 23–27. DOI: 10.1109/*ICSSP*.2012.6225976
30. Humphrey WS. Introduction to the Team Software Process. Addison-Wesley Professional: Boston, 2000.
31. Curtis B. Substantiating programmer variability. *Proceedings of the IEEE* 1981; **69**(7). DOI: 10.1109/PROC.1981.12088
32. Wolverton RW. The cost of developing large-scale software. *IEEE Transactions on Computers* 1975; **23**:615–636. DOI: http://doi.ieeecomputersociety.org/10.1109/T-C.1974.224002
33. Shadish W, Cook T, Campbell D. Experimental and Quasi-Experimental Designs for Generalized Causal Inference. Houghton- Mifflin: Boston, 2002.
34. Albrecht AJ. Measuring application development productivity. Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium, 8392. IBM Corporation: Monterey, California, October 1979. DOI: 10.1109/WETSoM.2012.622986
35. Corona E, Marchesi M, Barabino G, Grechi D, Piccinno L. Size estimation of Web applications through Web CMF Object. Proc. of 3rd International Workshop on Emerging Trends in Software Metrics (WETSoM). ICSE 2012: Zurich, Switzerland, 3 June 2012; 14–20. DOI: 10.1109/WETSoM.2012.6226986

AUTHORS' BIOGRAPHIES

Giulio Concas received the Laurea degree in Computer Science from the University of Pisa in 1989. From 1991 to 1999 he was graduate technician, in charge of the Scientific Calculus Area of IT Services Center of University of Cagliari. From 1995 he has performed several advanced consultancy activities in the field of computer science. He is a member of Local Linux User Group and an open source supporter. Since 1999 he is assistant professor of theory and applications of the Internet at the University of Cagliari. He has published more than 50 research papers in international journals and conferences. His research interest include Internet technology, service oriented architectures, open source software, agile methodologies and software engineering.

Maria Ilaria Lunesu received the Laurea degree in electronic engineering from the University of Cagliari, and the PhD from the same university in 2013. She works at the Department of Electrical and Electronics Engineering of the same University. Her research area is software engineering, in particular, software process simulation focusing on modelling and simulating maintenance and development process in the Agile and Lean context. More information about her can be found at her webpage at: http://agile.diee.unica.it/it/persone/29.html

Michele Marchesi received the Laurea degrees in electronic engineering and in mathematics from the University of Genoa in 1975 and 1980, respectively. He is professor of software engineering at the University of Cagliari. His research interests include software modeling using complex system approach, agile methodologies, open source development and applications, modeling and simulation of financial markets and economic systems using heterogeneous interacting agents. He has published more than 200 papers in international journals, books and conferences. He has been the leader of several research projects amounting various million Euros, and is a consultant for various companies and public bodies. He is member of IEEE.

Hongyu Zhang is an Associate Professor at School of Software, Tsinghua University, Beijing, China. Before joining Tsinghua, he was a lecturer at RMIT University, Australia, and a research fellow at National University of Singapore. His research area is software engineering, in particular, software quality engineering, metrics, maintenance, and reuse. His research focuses on improving software quality and productivity. He has published more than 60 research papers in international journals and conferences. He is a principal investigator of many national and university research projects. More information about him can be found at his webpage at: https://sites.google.com/site/hongyujohn