Ph.D. in Electronic and Computer Engineering
Dept. of Electrical and Electronic Engineering
University of Cagliari

Process Software Simulation Model of Lean-Kanban Approach

Maria Ilaria Lunesu

Advisor: Prof. Michele Marchesi
Curriculum: ING-INF/05 SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

XXV Cycle
April 2013

Ph.D. in Electronic and Computer Engineering
Dept. of Electrical and Electronic Engineering
University of Cagliari

Process Software Simulation Model of Lean-Kanban Approach

Maria Ilaria Lunesu

Advisor: Prof. Michele Marchesi
Curriculum: ING-INF/05 SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

XXV Cycle
Aprile 2013

Dedicated to Nonno GiovannAntonio

and

to My Family

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Lean software development [34] is a relatively new entry within the Agile Methodologies realm, but it is currently one of the fastest growing approaches among software professionals.

Lean is derived by Lean Manufacturing [1], which strives to maximize the value produced by an organization and delivered to the customer. This is accomplished by reducing waste, controlling variability, maximizing the flow of delivered software, focusing on the whole process, and not on local improvements, all within a culture of continuous improvement.

In 2003, the Poppendiecks published a book about Lean Software Development (LSD), applying lean principles to software production [1]. They identified seven key lean principles: eliminate waste[1], build quality in, create knowledge, defer commitment, fast deliver, respect people and optimize the whole [38]. In particular is the Kanban system [3] the latest trend in Lean software development, emphasizing a visual approach to maximize flow and spotting bottlenecks and other kinds of issues.

Recently, the Lean approach and concepts are becoming popular among software engineers because they are simple, structured and yet powerful.

Kanban processes have already been extensively studied through simulation in the manufacturing context [6]. The Lean approach is focused on maximizing the customer value and on reducing waste at every level. One of the key practices of Lean is to minimize the Work-In-Progress (WIP), it is the items that are worked on by the team at any given time. In LSD, an important tool for managing workflows and controlling waste, is the concept of "pull system" [1]: 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 way, developers are maximally focused, and the waste of time and resources due to switching, from one item to the other, 1 is minimized. A tool to minimize WIP - and to increase information flow about the project state 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 [17].

A Kanban board graphically shows the activities of the process in its columns. The items under work are represented by cards applied on the board in the column representing the

[1] 1 Waste in software development: partially done work; extraprocesses; extra features; task switching; waiting; defects [34])

item's current state. Each activity (column) has a maximum number of items that can be pulled into it, according to WIP-limits. 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 average time to complete one task).

One of the key concepts of Kanban system is that the whole development has to be optimized, avoiding local optima and striving to find global optima. The process itself is based on several parameters, and a global optimization is difficult. Moreover, these parameters are not fixed, but depend on factors such as the number and skills of developers, the specific practices used by the team, the number and effort of the features to be implemented.

These factors are not only variable among different projects, but can also vary during a given project. For these reasons, an up-front, one size fits all study and optimization of process parameters is out of discussion.

This is a typical problem where the simulation of the software process might be useful. In fact, the Kanban system is a very structured approach, and it is possible to design a software process simulator that captures most of what happens during development.

Recently, Ikonen et al. [41] 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.

Simulation is a generic term which includes a set of methods and applications that are able to represent how real systems work, typically using a general-purpose computer. Simulation has been applied to software process for more than twenty years.

A process for the development of softwares consists of a well-defined set of activities that lead to the creation of a software product. In literature there are many software processes different from each other but whose essential purpose is to be able to meet the expectations of customers, and above all, provide quality software products on time and within budget.

Many software processes have been adopted in industrial practices. For example, the Personal Software Process (PSP) [14], proposed by SEI, shows software engineers how to plan, track and analyze their work. The Team Software Process (TSP) [2] is built on the PSP and extends it to a software project team. Scrum [42] is an iterative, incremental software process, which is by far the most popular Agile development process [44].

Therefore, all software processes are driven by a common goal, but they differ from the modalities used to achieve these objectives.

In general its approach, toward the production of the software, could change, in fact the processes could be guided by the documents, by the code itself, or by risk, and so forth.

Today, the software companies are facing with one of the big problems typically a software process is envolving: the management of changing requirements and, as mentioned earlier, the ever-increasing demands by the market to be able to produce quality software in time and with reduced costs. The technical process simulation software, as it is considered now, in academic and industrial research, is a mean of assessing a software development process, and can help companies to manage certain changes in the development process and therefore is a valid support to the decision-making process.

They can be used specifically to help companies and manage the development process and also the changes that the reality of the software imposes. With the term simulation means the activity of replicate by means of suitable models a reality already existing or to be designed in order to study, in the first case, the effects of possible actions or somehow predictable events, and in the second case, evaluate various alternative design choices.

Compared to the direct experimentation, expensive and often virtually impossible, or

that achieved by means of mathematical models, the simulation shows the advantage of great versatility, the speed of realization and the relative low cost. And through the simulation it is also possible to explore and evaluate new management policies, design choices and alternatives model systems of great complexity by studying the behavior and evolution over time.

In general, the starting point in a simulation project, is the identification of a reality that will be analyzed in order to highlight inside the system which is going to be studied. The system is not a reality, but it is a representation, where the level of detail will depend on the objectives of the study and the type of problem that must be resolved. At this point one passes to the construction of a formal model that allows to simulate the identified system, in order to understand the behavior.

The simulation also allows a saving of time in the analysis systems: in fact, if one wanted to test the behavior of a real system, the analysis time would be very expatiated, while with a simulation model it will be employed a minimum time to have an overview of the whole system. Thanks to the simulation that can then be used in all these areas, we can start only from an abstract knowledge of the software system to be produced, and it happens without having yet determined how many resources need to be used as a model for the production process and how long it will take to make the product software.

For this purpose comes the idea of making a simulation model capable of reproducing different types of process and related approaches to various methodologies.

We developed a simulator of the Kanban process a WIP limited pull system visualized by the Kanban board. The simulator is fully object-oriented, and its design model reflects the objects of the Lean software development domain. We used this simulator, firstly, to assess comparatively WIP-limited and unlimited processes. We studied the optimum values of the working item limits in the activities, using a paradigmatic case of 4 activities and 100 work items. We performed an exhaustive search on all the admissible values of the solution, finding sensible optimal values, and a non-trivial behavior of the cost function in the optimization space.

This demonstrates the feasibility and usefulness of the approach. Nevertheless, in last years, the Lean software development approach has been applied also to software practice including software maintenance. Maintenance, is an Agile approach whose underlying concept is borrowed from manufacturing [35],it is also an important stage of software life cycle, which accounts for a large percentage of a system's total cost. After a software product has been 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 has often to cope with a long stream of requests that must be examined, answered to, and verified[47].

In practice, these activities are often performed with a no specified process, or using "heavyweight" processes [34] 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. The simulator we implemented, that is event-driven and agent-based has been used to simulate the PSP, the Scrum and the Lean-Kanban processes for software maintenance activities. One of the aim of this thesis is to demonstrate that the proposed simulation model, that implement a WIP-limited approach such as Lean-Kanban, can indeed improve maintenance throughput and improve work efficiency.

The proposed simulation-based method, as advocated by a Lean-Kanban approach, could help demonstrate the efficiency of a WIP-limited software maintenance process. Therefore

at first, we developed a generic simulator, we tuned it to reflect the original maintenance process, then we imposed a limit on the number of maintenance requests (i.e., WIP) a maintenance team can work on at each given time and used the tuned simulation model to simulate a WIP-limited maintenance process. The proposed simulation study showed that the WIP-limited process can lead to improvement in throughput. Furthermore, the WIP-limited process outperforms the original process that does not use a WIP- limit.

The scope to build this software process simulation model was also to reproduce post-release maintenance activities. Through simulations, we could show that the original process could be improved by introducing a WIP-limit. We assessed the capability of the model to reproduce real data and found that the WIP-limited process could be useful to increase the efficiency of the maintenance process. It has been shown that such simulation model, after a customization and calibration phase, can help to improve software maintenance process in industrial environments. In fact this process simulator, can simulate the existing maintenance process that does not use a WIP limit, as well as a maintenance process that adopt a WIP limit. We performed two case studies using real maintenance data collected from a Microsoft project and from a Chinese software firm.

The 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. In particular considering that software risk management is a crucial part of successful project management, but it is often an aspect not well implemented in real projects, we investigated the reasons. One reason is that project managers do not have effective tools and practice for the risk management of the software.

For that the Software Process Simulation Modeling (SPMS) is emerging as a promising approach to address a variety of issues related to software engineering, including risk management. Clearly, SPMS cannot address risk dimensions such as Organization environment stability and management support, lack of User involvement, Team motivation and communication issues. However, it can be useful to establish the impact of risks such as Requirement estimation errors, rework due to software that does not meet its intended functions and performance requirements (software technical risk), Project complexity, Planning and control practices. Nevertheless, in the present state of things, it should be better clarified how and to which extent SPMS can support software risk management, especially in the case of Agile methodologies. At this point we proposed a Monte Carlo stochastic approach to perform risk assessment of agile processes.

The main idea of our work was to model the process, the project requirements and the team characteristics, to identify the possible variations in model parameters that can cause any risk, to identify the key outputs to monitor, and to perform many simulations varying the parameters. The resulting distributions of key outputs can give an idea of expected outputs and of their variance, including proper risk percentiles. We performed an analysis of risk management in the case of a medium-sized project carried forward by a team of seven developers, with different skills.

We analyzed different variation rates both in term of estimation errors and of need of rework. We compared two agile approaches: one based on time-boxed iterations (Scrum), the other one based on continuous flow with work in progress (WIP) limits (Lean-Kanban). In both cases, we assessed the risks regarding project duration variations, and regarding lead times and cycle times needed to fulfill the implementation of features requested by the user– that is the time needed to give value to the customer. In particular a by-product of the analysis is also a comparative assessment of Scrum and Lean-Kanban approaches on a typical

case using simulation.

In this thesis we presented, moreover, a risk assessment procedure based on process modeling and simulation, and tested it on two popular agile methods –Scrum and Lean-Kanban – using synthetically generated requirements. We shown how it is possible to run the simulator in a Monte Carlo fashion, varying the identified risk factors and statistically evaluating their effects on the critical outputs. The proposed approach is clearly relevant for project managers, who get a tool able to quantitatively evaluate the risks, provided that the process and the project's data are properly modeled. This is made possible by the relative simplicity of agile processes, that typically operate on atomic requirements – the features –through a sequence of activities, performed during time-boxed iterations, or in a continuous, WIP-Limited flow like with Kanban. Using the existing simulation model, customized to follow the Scrumban process, we are working on the analysis and the simulation regarding data collected from a real experiment conducted in the context of Software Factory Network. The experiment is regarding a distributed development project conducted by a team composed by two groups of skilled developers located in two different sites in Spain and in Finland respectively. Thesis Organization:

In *Chapter 2* we showed the related work and the existing empirical studies on simulation modelling of Lean-Kanban aimed to study, the applications and the effectiveness of its practices, highlighting the main aspects faced in this work thesis, in particular simulation of development and maintenance processes, simulation considering a risk analysis and a hint regarding simulation of distributed software development;

In *Chapter 3* we reported an overview regarding Lean-Kanban approach and its practices and tools, we mentioned the potential of the use of Kanban approach;

In *Chapter 4* the main aspects of simulation and modelling and its advantages and disadvantages are shown in detail, we mentioned the main used models and the different approaches;

In *Chapter 5* we presented a description of the different actors of the model we developed and tested on different cases studies, for each actor the abilities and characteristics are shown;

In *Chapter 6* we presented different case studies that showed the applications of the simulator to real cases. For each case study a description and the calibration of the model are explained in detail underlying the different calibrations;

In *Chapter 7* we presented the experimental results that come from the simulation runs performed in the different case studies, in particular, we highlighted the results from the comparison among Scrum, Kanban and generic maintenance processes;

In *Chapter 8* we briefly discuss about the results obtained applying the proposed model to different real cases, reported in the previous chapter;

In *Chapter 9* we presented some threats to validity related to the different aspects of this thesis;

In *Chapter 10* we presented the final considerations related to the approach and the obtained results, finally we concluded the thesis works and we presented some future work.

Chapter 2

Related Work

The Lean approach, first introduced in manufacturing in Japan between 1948 and 1975 [15], strives to deliver value to the customer more efficiently by designing out overburden and inconsistency, and by finding and eliminating waste (the impediments to productivity and quality).

This is expressed in a set of principles that include optimizing flow, increasing efficiency, decreasing waste, improving learning, and using empirical methods to take decisions. In 2003, Mary and Tom Poppendieck published the first book about applying Lean principles to software development [1]. They identified seven key lean principles: eliminate waste[1], build quality in, create knowledge, defer commitment, deliver fast, respect people and optimize the whole. A key Lean activity is to build a value stream map (as reported in[1] pag.9) , breaking down the process into individual steps, and identifying which steps add value and which steps do not, thus adding to the waste. Then, the goal is to eliminate the waste and improve the value-added steps.

An important conceptual tool to manage how work flows is the concept of pull system (as reported in[1] pag.71), where processes are based on customer demand. The work in process (WIP) is usually made evident to the team, and to the stakeholders, using a Kanban board. In general, it can define the Kanban software process as a WIP limited pull system visualized by the Kanban board .

Recently, the Kanban approach applied to software development, seem to be one of the hottest topics of Lean. In the recent 3-4 years, Kanban has been applied to software process, and is becoming the key Lean practice in this field. A correct use of the Kanban board helps to minimize WIP, to highlight the constraints, and to coordinate the team work.

However, Lean is more than Kanban, and more Lean practices should be used, together with Kanban, to take full advantage of the application of Lean to software development. Note that the Kanban board is similar to the *information radiators* of Agile methodologies [16], but it is not the same thing.

To be eligible to use the Kanban approach, the software development must satisfy the two Corey Ladas' postulates [17]: Kanban processes have already been extensively studied through simulation in the manufacturing context [2]. Here we quote the seminal work by Huang, Rees and Taylor [7], who assessed the issues in adapting Japanese JIT techniques to

[1]Waste in software development: partially done work; extraprocesses; extra features; task switching; waiting; defects [1]

American firms using network discrete simulation. Among the many authors who studied Kanban using simulation we may quote Hurrion, who performed process optimization using simulation and neural networks [8], the data-driven simulator KaSimIR by Kchel, and

Nielnder, developed using an object-oriented approach [9], the hybrid simulator by Hao and Shen [10]. The simulation approach has been also used to simulate agile development processes. As regards Lean-Kanban software development, the authors are only aware of a Petri-net based simulation cited by Corey Ladas in his website [13].

The mathematical formalism of Petri nets, introduced to describe state changes in dynamic systems, is simple, yet powerful. However, it is not flexible enough to describe the details of software development, including features, activities, developers, deliverables. Using simulation optimization, the practice of linking an optimization method with a simulation model to determine appropriate settings of certain input parameters is possible to maximize the performance of the simulated system [14].

The growing interest on Kanban software development is demonstrated by the publication of various books, and by the proliferation of Web sites on the subject in the past couple of years. The most popular among these book was written by David J. Anderson [3]. Another book by Corey Ladas is about the fusion of Scrum and Kanban practices [17]. A third book on the subject was written by Kniberg and Skarin [18], and is also availabile online. The reader interested to a more detailed description of the Kanban approach should refer to these books. A well-defined software process can help a software organization achieve good and consistent productivity, and is important for the organizationâs 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) [60] proposed by SEI shows software engineers how to plan, track and analyze their work. The Team Software Process (TSP) [60] is built on the PSP and extends it to a software project team. Scrum [63] is an iterative, incremental software process, which is by far the most popular Agile development process [65].

In recent years, the Lean-Kanban approach [3] 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. 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 [46] proposed methods for simulating and analyzing software maintenance process. Otero et al. [61] use simulation to optimize resource allocation and the training time required for engineers and other personnel. In a previous work [51], 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.

Typically many software projects have suffered from several kinds of problems, such as cost overruns, lengthening of the time of delivery and poor product quality. One of the factors that cause these problems is the fact that the risks are not handled, as shown by Charette [78].

Risk management in software projects is a key component of the success of a project. Software engineering researchers and professionals have proposed a number of systematic approaches and techniques for effective risk management as reported by Boehm [79]. A

study conducted by the Project Management Institute has shown that risk management is an activity not practiced among all the disciplines of project management in the IT industry [80]. In real software projects, the risks are often managed using the insights of project manager, and the entire process of risk management is rarely followed [81]. One of the main reasons for this is that project managers lack practical techniques and tools to effectively manage the risks. Process Modeling Simulation Software (SPMS) is presented as a promising approach suitable to address various kind of issues in software engineering [4]. The results of the review conducted by Zhang et al. [5] showed that the risk management is one of the several purposes for SPMS. Liu et. al. performed a systematic review on this topic, concluding that the number of SPSM studies on software risk management has been increasing gradually in recent years, and that discrete-event simulation and system dynamics are two most popular simulation paradigms, while hybrid simulation methods are more and more widely used [82].

Regarding the application of SPMS to agile methodologies, system dynamics models were used in several investigations. Among them, we quote the paper of Cocco et al., who analyzed the dynamic behavior of the adoption of Kanban and Scrum, to assess their relative benefits [86]; this paper includes also some references to previous work on the topic. Cao et al. performed an extensive study of the complex interdependencies among the variety of practices used in agile development using system dynamics [84].

Discrete-event simulation of agile development practices were introduced by Melis et al. [11] [12]. Anderson and the authors of this paper proposed an event-driven, agent-based simulation model for Lean-Kanban process, extensible to other agile software processes, and used it to demonstrate the effectiveness of a WIP-Limited approach, and to optimize the WIP limits in the various activities [50]. In a subsequent work, Anderson et al. used an extended version of this simulator to compare Lean-Kanban with traditional and Scrum approaches on the data collected from a Microsoft maintenance project, showing that the Lean-Kanban approach is superior to the others [51]. Turner et al. worked on modeling and simulation of Kanban processes in Systems Engineering [52] [53]. 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 task and the accumulation of value, while Agent-based simulation is used to model workflow at a lower level, including working teams, Kanban boards, work items and activities.

Lamersdof et al. present a model for identifying risks at the beginning of a project [72]. The model systematically captures experiences from past projects in a set of logical rules describing how project characteristics influence typical risks in distributed development. Analysing the literature on distributed or global software development (GSD) they found risks of the use of distributed development vs. collocated software development.

Hawthorne and Dewayne [39] discuss the challenges and opportunities that software engineers face as a direct result of outsourcing and other distributed development approaches that are increasingly being utilized by industry, and some of the key ways they need to evolve software engineering curricula to address the challenges. They used an analysis of different interesting characteristics of distributed development. The differences from our work are about the fact that they try to identify best practices to help the software engineers to ongoing distributed development considering as important aspect the use of an architecture.

By Lundell et al. [40] an analysis of some OSS (Open Source Software) development model with distributed nature of a problem and the reasons to consider distributed development is offered . The goal of this work is to compare OSS development model with traditional

distributed development models, using, as starting point the caracterization of the context in which OSS development takes place. They use two frameworks to make an observation on how some practices relates to the issues DD work. These frameworks are populated like a matrix and their dimensions are: temporal, geographic, socio-cultural distances in one axis and control, coordination and communication in the other. The differences with our work are that they compare OSS with traditional distributed development model checking in the literature the role of the developer also as user.

Walkeland et al. [42] propose a general case for solving problems related to the a software development project by combining a service approach to system engineering with a Kanban-based scheduling system. The main goal of this work is, through modeling component, to verify whether organizing project as a set of cooperating kanban (a kanban-based scheduling system, KSS) results in better project performance. They used the combination of three different approaches to model the process System Dynamics, Discrete-event and Agent based. This work provides the basis for validating the approach with agent based simulation meanwhile we considered an event driven and agent-based approach to model and simulate the process followed and make a comparison with other processes.

Zhao and Osterweil [37] present the definition of a process for performing rework and a tool that executes the process in order to support human seeking help in being sure that they are carrying out rework completely and correctly. The focus of this work has been to present notions about rework with their ideas and conceptual basis, using a detailed specification of the process. They use a specific tool and made the comparison with other tools. In this work the authors analyzed the rework and its phases in detail using a specific tool and giving an example about the characteristics and benefits instead in our work the rework is a part of the used process.

Cavrak et al. [73] offer an outline of the characteristics of an existent flexible project framework for conducting student project in a distributed environment. Based on data collected from a distributed environment, they describe the analysis of cultural differences highlighting the goal that is to present a flexible project framework for conducting and evaluating distributed student projects and underline the done analysis of collaboration links within distributed team. They used a framework to identify the roles in the project. It is different from our work because they considered a general framework for running distributed student projects and base their work on an extensive experience.

A report on a process transition from Scrum to Scrumban in one software company is presented by Nikitina et al. [83], it gives an account on the process transition, changes done to the development process undergoing the transition and the improvement achieved. The paper present a brief and clear description of Kanban e Scrum. The main goal of this work is to show the transition from a process to another. Authors use an action research method combining two phases Initial Study and Software Process Transition. The differences from our work is that they show the steps to change process instead we analyze the results that come from the process in order to calibrate the simulation model.

Huston [38] presents an explanation of the opportunity for continued growth of this field lies in the reciprocity between research and industrial practice. An agenda for advancing software process simulation though this reciprocity is offered. He shows the current prominence of SPS (Software Process Simulation) using some research question underlining some aspects. In comparison of our work the hurdles that could be faced in SPS are shown.

Turner et al. [52] [53] show the simulation of the process performance vs. traditional methods of sharing systems engineering services across projects, and whether the overall

value of the systems of systems over time is increased. They developed a general Kanban approach a specific Kanban-Based process for supporting SE in rapid response environments and simulated that process as well as traditional process to determine if there were gains in effectiveness and value. The main goal of this work is, through modeling component, to verify whether organizing project as a set of cooperating kanban (a kanban-based scheduling system, KSS) results in better project performance. They used three different approaches to model the process System Dynamics, Discrete-event and Agent based. Martin and Raffo [29] present a detailed hybrid model of a software development process currently in use at a major industrial developer. They show also that simulation models of the software development process can be used to evaluate potential process changes. The main difference from our work is the used approach and the goal, in fact their hybrid model allow to simulate the discrete activities within the context of an environment described by a system dynamics model.

Chapter 3

Lean-Kanban Approach

3.1 Lean Kanban Development

In this section some typical practices of "Lean" development applied to the software are described. The "Lean" approach was introduced in the mechanical industry by the Japanese, and in particular from Toyota in 80s, and has been very successful at international level in 90s. Its application to the software development, is mainly due to Mary and Tom Poppendieck, which in 2003 published the book "Lean Software Development: An Agile Toolkit", in which they described the principles and practices related to this approach. This book was, actually, very general, such as those of Agile Manifesto, the principles were shareable but generic; they presented a series of 22 "tool" very diversified. For this reason, developers preferred methodologies structured and precise, such as XP and Scrum.

More recently, "Kanban", was found, in Lean approach, by David J. Anderson [3]. In fact, "Kanban" was originally the Lean practice to view the processing status of the involved various entities.

With Anderson, it becomes a view control of the process, which highlights the workflows and bottlenecks. The Kanban approach, is complemented by concepts of Theory of Constraints proposed by Goldratt and of Lean approach and currently is having great success in Agile software.

LSD: Lean Software Development was the first attempt to apply the "Lean" or "Toyota approach", typical of the industrial production, to the software process. In fact, software development is a form of product development. Much of software we use was probably purchased as a product. Software that is not developed as a standalone product may be embedded in hardware, or it may be the essence of a game or a search capability. Some softwares, including much custom software, is developed as a part of a business process. Customer don't buy the software we develop. They buy games, word processors, search capability, a hardware device or a business process. In this sense, most of the useful software are embedded in something larger than its code.

The software development is just a subset of the overall product development. And thus, if we want to understand lean software development, we should discover what constitutes an excellent product development. The Toyota Production System and the Toyota Product Development System from the same underlying principles. The first step in implementing lean software development is to understand these underlying principles. It is based on seven

principles of "Lean":

1 *Eliminate waste*

2 *Amplify learning*

3 *Decide as late as possible*

4 *Deliver as fast as possible*

5 *Empower the team*

6 *Build integrity in - Create systems inherently cohesive*

7 *See the whole*

"Lean approach" offers 22 tools, each one linked to one of the principles. The 22 tools are shown and described below.

Eliminate waste

Tool 1: Seeing Waste
The seven source of wastes of software development are as follows:

1. Partially Done Work
2. Extra Processes
3. Extra Features
4. Task Switching
5. Waiting
6. Motion
7. Defects

We an also consider useless management activities, project tracking, unnecessary measurements, permissions.

Tool 2: Value Stream Mapping
It is possible to define, in a graphic fashion a value flow map, i.e. activities that bring value to the company, by contributing to the software production and release that is paid by the customer. This map shows the assets, with their time, and idle time or standby.
We show, with pictures, the differences between the case of development "waterfall" Fig 3.1 and an agile iterative development Fig 3.2.

Amplify learning
Software development is an activity project, and not production. Therefore it is very variable and uncertain at first, because it needs to check the alternatives, and because the risk degree is high. The quality software is related to the quality of the experience that the customer has with it.

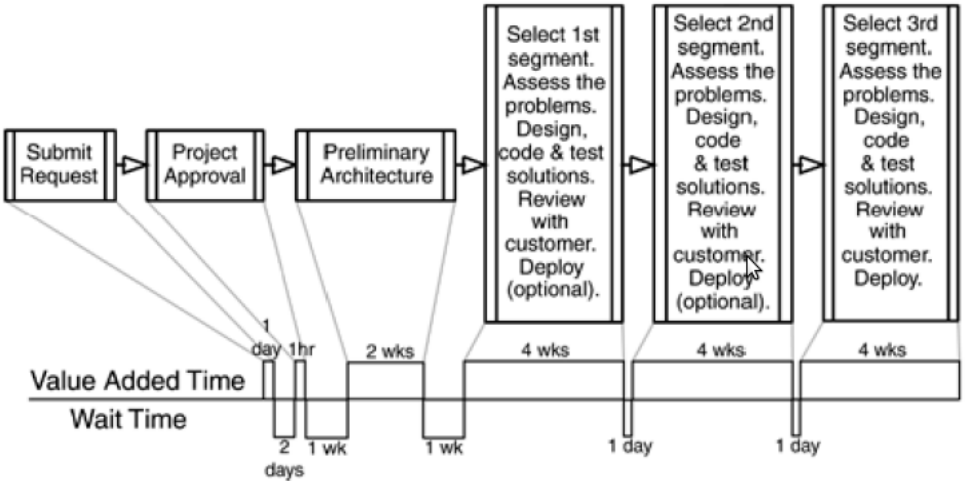

Figure 3.1: *Value Stream Mapping of a traditional software development.*

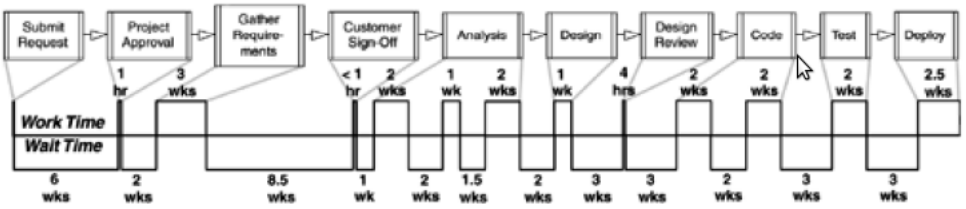

Figure 3.2: *Value Stream Mapping of software agile iterative development.*

A quality system has a received integrity (in the sense of consistency, completeness, ease of use), and a conceptual integrity. The integrity perceived by the customer is the balance of features, usability, robustness and economy that satisfies the customer. The conceptual integrity is related to social cohesion and good system design.

When designing software, it should, usually, use project cycles in which it matures and converges, and not try to "do everything right the first time." Of course, everything depends on the characteristics of the problem to be solved. Tools to extend learning are:

Tool 3: Feedback

Get feedback from the customer, the organization, by other developers, by the code, implies an iterative development. In general:

1. Do not let that defects accumulate, perform tests as soon as the code is written.
2. Instead of adding documentation, try new ideas by writing code.
3. Instead of collecting additional requirements by the user, show screens test and hear what he says.
4. Instead of studying in detail what tools to use, could you provide the three most po pular and try all of them.

5. Instead of trying to convert a legacy system all at once, provided to Web interface, in order to experiment with new ideas here.

Tool 4: Iterations

The iterations are the increase to the software that is designed, written, tested, integrated and released in a short and predetermined period of time. There are three basic principles related to the iteration:

1. Development for small batches, that improve the quality, communication among developers, resource utilization and feedback.

2. The iterations are a method based on the options, the choice may be delayed. The development is controlled based on the facts and not on the assumptions.

3. The iterations are points of control and synchronization within the team, with the customer and between any multiple teams.

At the beginning of each iteration there is a planning session between team and customer, or his representative, to decide which features to implement. The team evaluates the functionalities and says what can be implemented in the iteration. A team must get support from the organization: skills, information, resources, environment programming and integration.

The iterative approach can lead to non-convergence (thrashing), if the functionalities change too quickly. During the iteration, the team must be allowed to work. The iteration should not be too short, not to favor the trashing, nor too long,in order to maximize feedback.

It is ideal to have a "negotiable scope", in order to implement the most important features, leaving the team and customer to remove and add secondary features. The implemented features are displayed in a "burn-down" graph to keep under control the development and the convergence of the system. You can, also, use the graph of the number acceptance tests exceeded, compared to the number of those defined.

Tool 5: Synchronization

If the collective ownership of the code is used, it needs to integrate often in order to synchronize efforts. To properly integrate, it is good that the system is equipped with automated testing, which must all be passed before integration.

Another tool of synchronization is to immediately implement a vertical "slice", maybe simplified, of system functionality (Spanning Application), and then follow with the other features, developed by imitating and improving first.

Another approach to synchronization is the matrix, in which various teams develop components or subsystems in parallel, with synchronization points. Interfaces, and established control points must be defined before.

Tool 6: Set-Based Development

This instrument addresses the problem of multiple project constraints. Instead of declaring one at time, satisfy them and then, check if the new ones are compatible with the old, it is better to put them all on the table, giving the permissible ranges for each constraint (sets), and see if there is an intersection that can satisfy all of them.

The software can be developed in parallel with some different solutions to a problem, and then discuss them and use the best features of each one. It is important that each solution is used to communicate the constraints, not the specific solution.

Decide as late as possible

In software, the cost of correcting errors, or incorrect requirements, grows exponentially according to the correction done during analysis, design, coding, or after release. In fact, not all requirements have costs of correction that grow in the same way. The cost of correction of higher cost constraints (high-stake constraints) can have an exponential very steep.

The iterative and incremental development allows a delay of many design decisions, saving in the event of changes in constraints/requirements that occur after the start of the system.

Tool 7: Options Thinking

Each choice of implementation should be considered as if it were an option, with associated cost and benefits, to be calculated and to be taken into consideration. The various features can be associated with more alternative options. The choice of which option to exercise effectively be made as late as possible, when the costs and benefits of the alternatives are more stable and clearer.

Tool 8: The Last Responsible Moment

In particular in a parallel development of a system, the final decisions should be taken in the last reasonable instant, that is, when not taking them an important alternative is eliminated, that you can not take any more. This does not mean delaying the project. Some tactics are:

1. Share project information even if is partial.
2. Encourage direct collaboration among developers.
3. Develop a sense for how to absorb the changes. In practice:
   - use software modules;
   - use interfaces;
   - use parameterized modules;
   - use abstractions;
   - avoid sequential programming in favor of the declarative;
   - attention to the construction of frameworks and specific instruments;
   - avoid duplications;
   - separating concerns: Each module must have a single responsibility well defined (cohesion);
   - encapsulate changes, leaving the stable interface;
   - delay the implementation of future capabilities;
   - Avoid extra features.
4. Develop a sense for what is critically important in the domain.

5. Develop a sense for when they made the decisions.

6. Develop the ability to respond quickly.

Tool 9: Making Decisions

Development approach with "breadth first" (attempting to cover the whole system, low level of detail, then increasing it across the width) as opposed to the "depth first" (it could be covered in detail a portion of the system, then another, and so on). In general, it is better the first. Intuitive decision making opposed to rational decisions.

Small teams able to self-organize. Swarm intelligence:

- flexibility: the team adapts quickly to changes in the environment;
- strength: even if one or more members leave, the group can still operate well;
- Self-organization: the team requires minimal supervision.
- Use simple rules but clear and followed by all. I.e., The 7 principles of "lean."
- Delivered as quickly as possible

From the initial idea to the finished product should spend as little time as possible, but without hurry and compromising the quality.

Tool 10: Pull Systems

Developers must always have something to do, and have to know by themselves, without someone tell him. It needs to avoid the "micromanagement". In the "pull", who works at a processing stage starts only if the previous step ends and passes another product. The products state being processed, including those from to start and finished products for each phase are displayed in a board, with the "Kanban" approach (see fig 3.3).

The processing unit is the "Feature", or functionality, that constitutes the requirements of the system. The stages can be eg.: Analysis, coding, testing, integration. Each feature corresponds to a card, which is put on a board and that shows its status (Fig. 3.3).

In addition to the cards and the board, it should be done a daily meeting of 15' in which the team exchange information about what each of them has done and will do. Furthermore, in addition to the board Kanban, it is well make use of other "information radiators" to share problems, ideas, candidates for refactoring, etc.

Tool 11: Queuing Theory

The queuing theory is the transit time and waiting for elements in the queue. The cycle time (Cycle time) is the time it takes to travel an entire queue, including any initial waiting. It should be reduced to the minimum. There are two ways to reduce the cycle time: reducing the arrival rate in the queue, and reduce the transit time within the queue.

In input should be avoided arrivals blocks of features too large. It is better to divide the work in basic functionalities, that are distributed over time. This also facilitates the parallelization of their implementation. More details on the management of queues in the Theory of Constraints.

Tool 12: Cost of Delay (Cost-benefit analysis)

It needs to well know how to calculate economic implications of delays in delivery,so as other design choices and development. A delay may, for example, help to lose a window of opportunity with lost earnings much higher than the simple cost of delay in itself.

Each project should have an associated analysis, kept up to date, its costs and benefits present and future, in order to assess the impact of decisions. All costs must be quantified in dollars, or Euros, to be compared. See the examples in the book.

Empowering the team

The Lean approach than Fordism, ie the so-called "scientific approach" to planning industry, in which people are guided in detail by their leaders. The assumptions of CMMI, Fordist approach to the production of software are:

1. The best way to manage a system is break it down into modules to produce an easily defined way, which are transformed from one state of inputs to one output in order to obtain specific purposes.

2. In a mature organization everything is carefully planned and then checked for meet the plan.

The Lean approach, however, is based on alternative assumptions:

1. Mature organization considers the system as a whole and does not focus on optimize disintegrated.

2. Mature organization focuses on effective learning, and gives people who do the work the power to make decisions.

Tool 13: Self-Determination

The power to make decisions should be given to the people who work. A Work-out is a meeting of a few dozen developers, which can takes 2-3 days, in order to present proposals on how to improve the work. The manager must decide, within one month, which proposals to accept. Applicants of accepted proposals must, immediately, begin their implementation.

An idea for a manager or project leader is to treat employees as if they were a group of volunteers.

Tool 14: Motivation

If people have a purpose, usually seek to pursue it with care. If people do not have one purpose, they tend to get confused and uncomfortable. To give a sense of purpose to a team:

- Start with a motivation (a vision) clear and compelling.
- Make sure that the goal is achievable.
- Give the team access to customers.
- Let the team take their commitments alone.
- The manager's role is to listen, provide support and protection from interference external.

- Keep away from the skeptics team.

Each type of developers should have their own purposes. The building blocks of the reasons are:

1. Membership in the group, which includes mutual respect and reduced international competitiveness.
2. Safety of the workplace and the fact not to be punished in case of errors.
3. Expertise with a disciplined environment, professional and with all the tools necessary.
4. Progress: it needs to give a sense of achievements, perhaps the most celebrated important. Iterative development ease that.

Work more than eight hours can be a sign of a motivated and passionate team. But it is not sustainable in the long term, and also must be careful not to establish an atmosphere in where people are morally obliged to work overtime.

Tool 15: Leadership

There is a difference between managers and guide (leader). A manager faces the complexity and plans budgets, organization, tracking and control. A guide faces change: it gives the direction, aligning people, giving the reasons.

In software, it often happens that the development of large systems emerge a small group of exceptional designers who take responsibility for most of the project. In general in the team there are masters developers for competence and experience are the natural leaders of development.

Usually emerge spontaneously, but must then discover and use them in the best possible as an interface to customers and management. One of the tasks of the master developer, in iterative development, is to determine when the design of a system is sufficient to begin encoding as soon as possible, so as to get feedback and then to develop the project.

Tool 16: Expertise

Organization of software development requires both skills technological (analysis, languages, databases, user interfaces, Web technologies, ...) and of domain. To develop them, are very useful community competence (Communities of Expertise), in where experts exchange views and experiences directly or via the Web

And the task of the community also responsible for defining coding standards and development.

Create systems inherently cohesive

A successful system must have integrity perceived and conceptual integrity, and thus have integrity "built in". According to Bill Curtis, the three fundamental requirements for achieving this are:

1. High level of expertise in the application domain in the team.
2. Acceptance of change as the norm, and the ability to adapt to the changes in emerging project during construction.

3. An environment that fosters communication, integrating people, tools and information.

Tool 17: Perceived Integrity

The perceived integrity requires a strong focus on customer needs, and the use of feedback. To establish a good exchange of information with customers, there are several techniques:

- Small systems are developed by a single team, with short iterations and feedback from maximum number of potential users at each iteration.
- Extensive testing by customers are an excellent means of communication client-team.
- Complex systems are represented in a comprehensible language and simple models for users and that developers can use without further refinement.
- Large systems must have a master developer with deep knowledge of the customer and excellent technical credentials, whose role is to facilitate the emergence of project, representing the needs of the customer with the developers.

The Model Driven Design (MDD) could be used by developing analysis and design models using a understandable language to the user, and converts the code easily, and perhaps semi-automatic. In order to serve a set of models:

- A conceptual model of the domain, representing the classes with the basic entity of the system, or at least the basic entities and their relationships. The details must express key concepts, but do not be too refined.
- A glossary with terms of the model, in the model language.
- A model of the use cases, with descriptions of the scenarios of dynamic operation of the system.
- Qualifiers expressing estimates or constraints on the system, eg. the no. expected users, the response times acceptable, the rate of acceptable defects, etc..

Tool 18: Conceptual Integrity

If a system has conceptual integrity, all of its parts, at various levels, work together cleanly and consistently. When a new system is created, it is better that there are few new and innovative parts, reusing the other.

The conceptual integrity is not achieved the first time around, but with a long project work and iterations. For material goods, using the integrated problem solving, that follows the following principles:

1. Understanding of the problem and its solution must be carried out simultaneously, not sequentially.
2. Preliminary information is disseminated immediately. The flow of information between designers of high-level and developers is not delayed until these are not complete.
3. The information is frequently transmitted in small sets, and not all at once.

4. The information flows in two directions, not just one.
5. The best way to transmit information is the face to face communication, and no documents or e-mail.

The software usually has multiple layers:

1. Presentation (UI).
2. Services (transaction management) in many systems is omitted.
3. Domain (business logic).
4. Translation (mappings, wrapper): in many systems is omitted.
5. Sources of data (persistence, messaging)

The lower layers should not depend on the top ones. The way to obtain the conceptual integrity in the software is similar to that of physical products:

- Reusing parts and existing approaches: for example, wrappers to access legacy databases, XML and XML parsers available, adherence to standards.
- Use the integrated problem solving.
- Ensure that critical areas are addressed by experienced developers.
- Complex systems require the guidance of a master developer.

Tool 19: Refactoring

It is the notorious practice of XP. Refactoring is typically done in small steps. After each small step, you're left with a working system that's functionally unchanged. Practitioners typically interleave bug fixes and feature additions between these steps. So refactoring doesn't preclude changing functionality, it just says that it's a different activity from rearranging code.

The key insight is that it's easier to rearrange the code correctly if you don't simultaneously try to change its functionality. The secondary insight is that it's easier to change functionality when you have clean (refactored) code.

Refactoring is a kind of reorganization. Technically, it comes from mathematics when you factor an expression into an equivalence - the factors are a cleaner ways of expressing the same statement. Refactoring implies equivalence; the beginning and end products must be functionally identical. You can view refactoring as a special case of reworking.

Practically, refactoring means making code clearer and cleaner and simpler and elegant. Or, in other words, clean up after yourself when you code. Examples would run the range from renaming a variable to introducing a method into a third-party class that you don't have source for.

Refactoring is not rewriting, although many people think they are the same. There are many good reasons to distinguish them, such as regression test requirements and knowledge of system functionality. The technical difference between the two is that refactoring, as it was stated above, doesn't change the functionality (or information content) of the system whereas rewriting does. Rewriting is reworking.

Tool 20: Testing

XP testing was different in many ways from "traditional" testing. The biggest difference is that on an XP project, the entire development team takes responsibility for quality. This means the whole team is responsible for all testing tasks, including acceptance test automation. When testers and programmers work together, the approaches to test automation can be pretty creative!

As Ron Jeffries says, XP is not about "roles", it's about a tight integration of skills and behaviors. Testing is an integrated activity on an XP team. The development team needs continual feedback, with the customer expressing their needs in terms of tests, and programmers expressing design and code in terms of tests. On an XP team, the testers will play both the customer and programmer "roles". They'll focus on acceptance testing and work to transfer her testing and quality assurance skills to the rest of the team. Here are some activities testers perform on XP teams.

- Negotiate quality with the customer
- Clarify stories, flush out hidden assumptions
- Enable accurate estimates for both programming and testing tasks
- Make sure the acceptance tests verify the quality specified by the customer
- Help the team automate tests
- Help the team produce testable code
- Form an integral part of the continuous feedback loop that keeps the team on track.

The biggest difference between XP projects and most "traditional" software development projects is the concept of test-driven development. With XP, every chunk of code is covered by unit tests, which must all pass all the time. The absence of unit-level and regression bugs means that testers actually get to focus on their job: making sure the code does what the customer wanted. The acceptance tests define the level of quality the customer has specified (and paid for!)

Testers who are new to XP should keep in mind the XP values: communication, simplicity, feedback and courage. Courage may be the most important. As a tester, the idea of writing, automating and executing tests in speedy two or three week iterations, without the benefit of traditional requirements documents, can be daunting.

Testers need courage to let the customers make mistakes and learn from them. They need courage to determine the minimum testing that will prove the successful completion of a story. They need courage to ask their teammates to pair for test automation. They need courage to remind the team that we are all responsible for quality and testing. To bolster this courage, testers on XP teams should remind themselves that an XP tester is never alone.

Consider the whole

The "Systems Thinking" is an approach that looks at organizations as systems, analyzing how its parts are connected and how the organization behaves as a whole. It has main patterns

1. Limits to Growth: Even if a process or approach leads to the desired results, usually creates effects that outweigh these results. If the process is pushed further to increase the results, the secondary effects grow more, and the growth is limited. It should be instead find and remove limits to growth.

2. Move the load: if a problem cause side effects, but it is difficult to deal with, it could tends to treat the effects instead. However treating the symptoms and not the cause, the problem usually worsens.

3. Sub optimization: there is often a temptation to divide a complex system into smaller simple sections, and manage them as if they were independent. Improving the individual parts, however, is not said it best everything.

Tool 21: Measurements

The measures of process, product, cost, etc.. occur at the local level, but to use them to optimize local performance it is often counterproductive for a global optimization. It needs to sub-optimize for several reasons:

- Superstition: it is a not established association of cause and effect.

- Custom: to act and measure because "it has always been done in this way", and not for good reasons.

The measure of performance is often unsuitable, when it comes to knowledge workers. Typically, we try to cover all aspects with the approach (wrong!):

1. Standardization: it abstracts the process of development in sequential phases, prescribing how the various stages must be made, and it is measured according to the process.

2. Specification: to create a detailed development plan, and measuring performance against floor.

3. Breakdown: the great tasks are broken down into smaller tasks, which are in When measured.

The correct way to measure is, instead, to try to measure everything, and as much as possible in an aggregate fashion. You have to measure the information, not performance, aggregating to hide and mediate individual performance. Usually it makes little sense to assign the individual errors developers: they are a consequence of the system and development practices, not negligence of the individual.

Tool 22: Contracts

The Lean approach of Poppendieck devotes much space to the discussion and specific types of contract as possible regarding software development. In general, the contract at a predetermined price is not recommended, which locks the price and requirements, but means that the supplier adheres to them, and charged too much for any changes, and contracts based on man-hours (Time-and-Material) that are flexible to the requirements, but encourage the supplier to be inefficient. The contracts are recommended:

- Contracts Multistage (Multistage Contracts), used both to clarify in advance fixed-price contracts, which for the whole contract. Development occurs in a succession of contracts limited, with checks at the end of each one.
- Contracts with Cost Objective (Target-Cost Contracts), in which the total cost is responsibilities of both the customer and the supplier. It is a fixed price contract in which, if the goal of cost is exceeded, both parties lose out while, if it is centered, both sides earn. For example:
  - Cost plus fixed bonus: the price does not include the profit of the supplier, which is given only at the end of the project. If the actual cost exceeds the target cost, the supplier works in the cost price for the missing part. If the actual cost is less than the cost goal, the supplier has a greater profit.
  - Profit should not be exceeded: the price includes the profit, but if the real cost exceeds the target cost the vendor provides a cost price of the extra hours.
  - Agreements with objective of time (Target-Schedule Contracts): in the case where the finish work on time is more important that not exceed the cost, this type of contracts identifies functionality priority, and ancillary features, with the aim of ensure, however, that are completed within the stipulated time those priorities.
  - Contract with shared benefits (Shared-Benefit Contracts) are designed to promote full cooperation between customer and supplier because they give benefits both if the project is successful. An example are the contracts "co-source", in which both produce the software: half is produced by the team of the client, and half from the team of the supplier, working in cooperation, typically with an agile approach.

  All these contracts have in common the fact that not prescribe in advance the requirements of the system.

3.1.1 Development for feature

The premise of the approach ToC is that the work of software production is broken down into elementary units, implementable one at a time. Such is the agile approach, in which the requirements are divided into elementary units (User Stories, Features, Backlog items, as well as use cases). Elementary units called henceforth feature) evolve in stages: acquisition, analysis, design, coding, testing, integration. This allows to model the software process as a flow of processing feature.

It needs to estimate the importance of the features, in order to have a tool for their prioritization. The implementation flow can be drawn with a flow diagram, such as those used in Lean approach, shown in Fig 3.3. A vertical *section* of features under work shows the amount of *Work in Progress* (WIP), i.e. the number of features handled at the same time. An horizontal *section* gives the average time for processing a feature.

It is also important to note that the features are *perishable* in the sense that their value decreases average over time as the requirements tend to change and make obsolete

those defined in the past.

3.1.2 Kanban approach

In the literature there are some simulation models of the Lean-Kanban approach for manufacturing processes. For example, Huang et al.[7] assessed the issues in adapting Japanese *Just In Time* techniques to American firms using network discrete simulation. Hurrion [8]] performed process optimization using simulation and neural networks. Köchel and Nieländer [9] proposed a data-driven Kanban simulator called KaSimIR. Hao and Shen [10] 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. [11] [12]. A Petri-net based simulation of Lean-Kanban approach was cited by Corey Ladas in his website [13].

In a previous work we presented an event-driven simulator of the Kanban process, from which we derived the different versions of the simulator presented in this thesis. Using this simulator we could show the advantages of a WIP-limited approach versus a non-limited one, on synthetically generated data [50].

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 [51]. 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 [52] [53]. 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, while Agent-based simulation is used to model workflow at a lower level, including working teams, kanban boards, work items and activities.

Kanban means "visual signal" in Japanese. More than a development process is a set of practices that support Lean approach. Such practices, initially aimed at improving the exchange of information in a Lean team, then evolved into a more structured approach. The objective of the Kanban is to provide visual tools to facilitate the Lean approach, viewing and also providing constraints to minimize the Work in Progress (WIP) and highlight the bottlenecks. In this sense, Kanban is complementary to Lean and ToC. With Kanban:

1. It is fully displayed the workflow:
   * each feature is written on a card and placed on the board;
   * columns denote the state of the board cards in them posted.
2. The WIP is limited by limiting the number of cards that can be on different columns.

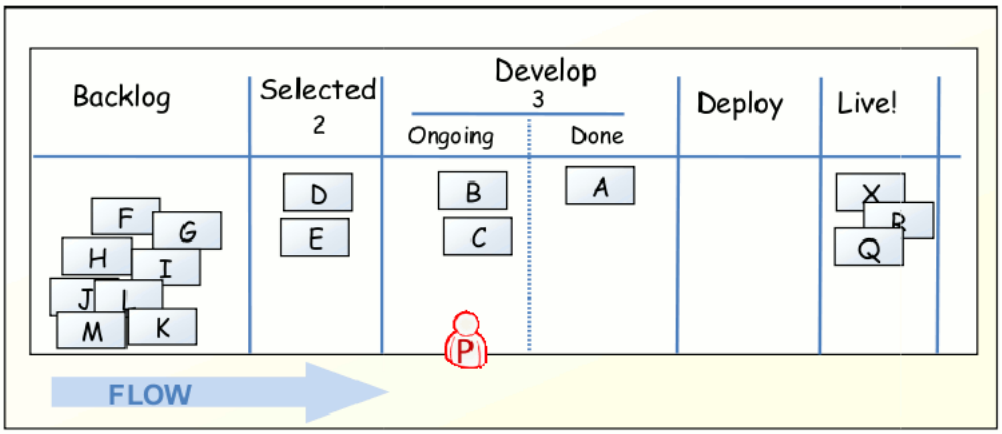

Figure 3.3: *A typical Kanban board.*

3. The lead time is measured (average time to complete a feature), to optimize the process making it less and less variable as possible.

In Fig 3.3 is shown a typical board Kanban, simplified. Each feature is to implement written on a card, in the figure denoted by a letter of the alphabet. The columns are following:

- Backlog: Here there are the accumulated feature to implement. Since this Kanban not explicitly contemplates the collection activity requirements (creation of the features), it is assume that these features are given as input uncontrolled development.
- Selected (2): here are located the selected features for implementation. At most there are 2. When one of them is shifted to the right because it starts developing, you can "pull" another by the total backlog. This is typically that of higher priority.
- Develop (3): the features under development, at most 3. Are in turn divided into:
  - Ongoing: feature under development;
  - Done: feature ready for release.

  The constraint on the 3 feature at the most, cumulatively on feature and Ongoing Done.
- Deploy: the feature being "deploy" (release). There are no constraints on their number.
- Live!: The feature release and running.

A more detailed and realistic Kanban board is shown in Figure 3.4, with the features obtained breaking up the "user stories".

The movement of the cards on the board can, easily, get the flow diagram described in ToC approach. Furthermore, the observation of the board shows just what activities are the bottlenecks. After each activity it is better if there is a column "buffer" in which to deposit the feature whose processing is complete, waiting to be "pulled" in subsequent processing [3].

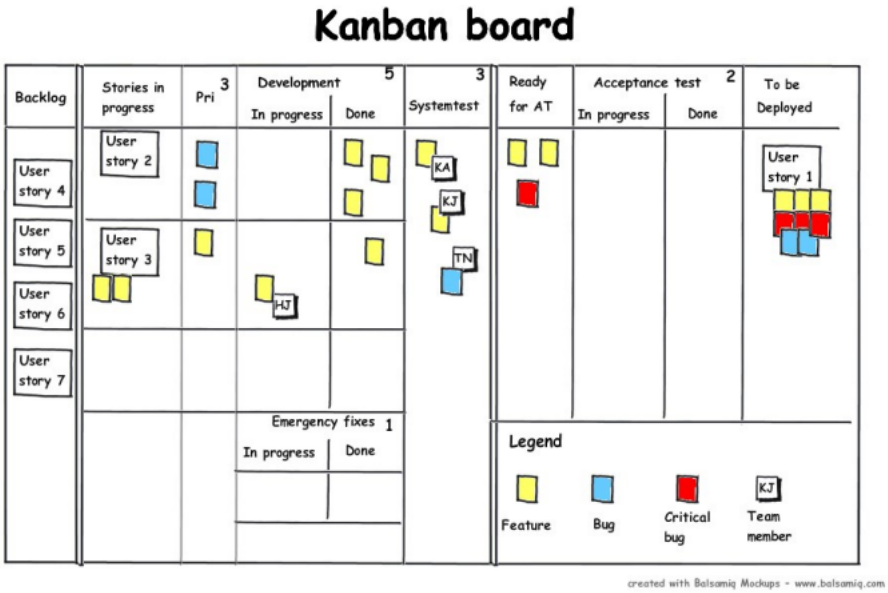

Figure 3.4: *A Kanban board more complex.*

Often the Kanban approach is used with Scrum, even if it involves a continuous flow implementation of the feature, which is taken from the Backlog whenever a vacancy in the active column. The use of "time boxed" iterations as in Scrum is therefore necessary any more, even if the activities of planning and review that occur at the beginning and end of each iteration must be disclosed with frequency, in order to maximize the communication and feedback.

It is also important the team management and individual team skills. Even in this case, there is no a specific approach, but using approaches borrowed from Scrum, XP or other agile methodologies.

Since the process of feature is made in specific activities, one type of activity could be carry by a specialist of at only one activity (eg. testing or design), or developers generalists able to choose the activities at that time more critical.

It is reported that the size of a team that follows a Kanban approach can also be greater than that of typical Scrum or XP team, and up to 50-70 people. The case of large teams, roles will typically be more specialized than in the case of teams of 5-10 people.

The choice of what are the specific features at some stage on which work is left to the developers, although it is advisable that a developer, once started the activity on a feature, bring it to at the end. The Kanban approach is compatible with the development of more projects in parallel, because individual feature may also belong to different projects. In this case, using cards of different color, or drawing horizontal "swim lanes" on the board, referring to the various projects.

A non-iterative process that follows the Kanban approach might consist of:

1. Preliminary activity of gathering requirements to determine and evaluate a set of features to start the development of the system.

2. Preliminary activity to determine the basic architecture of the system (see approach FDD).
3. Development by feature, with pull approach using the Kanban board. This development is asynchronous and iterative.
4. Regular meetings of retrospective verification of the work in progress, determination of the hills of bottle, improvements to the process.
5. Periodic meetings of planning, with collection of feedback from the user and revision / addition of features of the system.
6. Daily stand-up meeting.
7. Release management at pre-established times, with possible negotiation of the features planned for each release.

Further refinements can be the management of bug fixing (equivalent to a bug or feature to class of bugs connected), refactoring (also broken down into features, or left discretion of developers within the coding work of feature), functional test and acceptance (also similar to features).

Problems related to Lean and similar approaches The Lean approach the original, with the 7 principles of Lean is generic and very reasonable, and does not pose particular problems.

The approach ToC and / or Kanban assumes that the development is broken down into features, size comparable and the processing of which evolves according to a time sequence of activities (including possibly with cycles of rework).

Chapter 4

Software Process Simulation Modeling

Software Process Simulation (SPS) is a discipline with its own body of knowledge, theory, and research methodology. The main aspect of the discipline is the fundamental notion that models are approximations for the real-world. To engage Software Process Simulation, practitioners must first create a model approximating an event. The model is then followed by simulation phase, which allows for the repeated observation of the model.

After one or many simulations of the model, the analysis step takes place. Analysis aids in the ability to draw conclusions, verify and validate the research, and make recommendations based on various iterations or simulations of the model. These basic rules combined with visualization, the ability to represent data as a way to interface with the model, make Software Process Simulation a problem-based discipline that allows for repeated testing of a hypothesis.[70]

Software Process Simulation also serves as a tool or application that expands the ability to analyze and communicate new research or findings. In last years, we witnessed the diffusion and rise in popularity of Process Simulation. In fact, new and innovative software engineering techniques are constantly being developed, so a better understanding of these is useful for assessing their effectiveness and predicting possible problems. Simulation can provide information about these issues avoiding real world experimentation, which is both time and cost-intensive.

This area of SPS has attracted growing interest over the last twenty years, but only recently, is it beginning to be used to address several issues concerning the strategic management of software development and process improvement support. It can also help project managers and process engineers to plan changes in the development process. The development of a simulation model is a relatively inexpensive way compared to experimenting with actual software projects of gathering information when costs, risks and complexity of the real system are very high.

In order to relate the real world results to simulation results, it is usual to combine empirical findings and knowledge from real processes. In general, empirical data are used

to calibrate the model, and the results of the simulation process are used for planning, design and analyzing real experiments [71].

Likewise, to have a better understanding of this area, in particular what model and simulation model means, some key concepts have to be explained. A model is defined as an abstraction (i.e., a simplified representation) of a real or conceptual complex system. A model is designed to display significant features and characteristics of the system, which one wishes to study, predict, modify, or control. Thus a model includes some, but not all, aspects of the system being modeled.

A simulation model is a computerized model that possesses the characteristics described above and that represents some dynamics of the system or phenomenon. One of the main motivations for developing a simulation model or using any other modeling method is that it is an inexpensive way to gain important insights when the costs, risks, or logistics of manipulating the real system of interest are prohibitive. Simulations are generally employed when the complexity of the system being modeled is beyond what static models or other techniques can usefully represent.

4.1 What is a simulation model?

The process of the model construction is the starting point of the simualtion concept: using the simulation it is possible to test not only the actual production system that we would like to realize, but its abstraction. A model is then the abstraction of a real system throughout a set of data similar to it. In general, it is possible to build a model by isolating the important elements of the system and establishing meaningful relationships between them. In practice, each model is valid or not, depending on the specific objective that is pursued.

There are different kind of models, quantitative models, qualitative models, analytical, and in general it is usual to divide them into two main categories: static models and dynamic models. In particular, the static model is used to solve systems which are in equilibrium conditions (for example, problems relating to the structural calculation in engineering of structures), instead the dynamic model is used to solve systems that evolve over time and therefore where it comes into play the time as a variable.

The simulation models are generally dynamic models, which include, therefore, also the temporal dimension, and have the aim of studying the variation with time of a system.

A simulation model may be, for example, deterministic or stochastic, or discrete or continuous. It comes to deterministic simulation when the time evolution of the model constructed is uniquely determined by its characteristics and initial conditions. When, in the model, there are random variables that depending on the value taken can lead to different behaviors it is spoken of stochastic simulation. Continuous simulation includes a simulation in which the value of the involved variables varies continuously over time. It has instead a discrete simulation when the state of the studied system, and the value of the variables, changes in well defined instants of time.

In general, a simulation model is composed of a simulation algorithm, characterized by an input that consists of two types of data, those defined in the initial stage of set-

ting and which are not modifiable by the user, and the variables, user-editable , and which represent the possible situations that can be recreated at every simulation step. In output, instead, would have the results, which are starting point of the simulator itself, that is those results derived from decisions taken by the user for each simulation. These results are the ultimate goal of the simulator, in fact on the basis of the obtained results will be taken decisions also strategic.

Today, in fact, the software processes simulation models are used in different fields and from different points of view, because their ultimate goal should be to provide answers to different decision questions, and then are directed to the resolution of strategic issues such as software process management or as a support to the development process improvement. The process of creation and use of a simulation model consists of the following phases:

1 Formulation of the problem: it is necessary to define the objectives of the si mulation, and therefore what accurate information to get from the simulation, to do this it needs to establish what are the boundaries of the system to be studied. Make simplifications and assumptions, and then determine the right level of detail, it is essential for the success of the model.

2 Collecting and processing data: The collection of data relating to the reality to be represented is a long phase but essential for the success of a simulation model that really works. The data must be properly collected and analyzed in order to construct the variables.

3 Parameter setting: to define the values of the parameters that will act with the features of the model in order to determine the temporal evolution.

4 Construction of the model: the entities of the model and the functional relationships that link them are identified.

5 Coding of the computer model: This step consists of the translation of the model into a model can be interpreted by computer.

6 Validation of the model: the created model is then analyzed in detail as an abstraction of the real system, and eventually to understand the causes, resulting in errors and differences. Testing the model with a set of sample data and the results are compared. Therefore, the analysis of the behavior of the constructed model and the comparison with the initial data allows to understand how it represents, properly its objectives, and the reality being studied. If there is a good correspondence, one can consider the model validated.

4.2 Advantages of simulation

The simulations are generally employed when the complexity of the system to be modeled is such that a static model or other techniques are not useful for its representation. [4] Furthermore, since a model is an abstraction of a real system, it will be represented only by a few of the many aspects of a software process, which will be those considered to be basic to meet some goals.

The purpose of the construction of simulation models is precisely to be able to conduct virtual experiments and observe the system behavior under certain extreme conditions, or what happens with changes of some parameters. In order to obtain accurate results, it needs to perform many simulation runs using the model before collecting any data of a statistical nature.

Furthermore, the experiments that can be conducted are multiple, essentially of two types: an interactive approach and a comparative ones. In the first case, running the model it will observe what happens, for example, an action is implemented and it can be seen, how with this change, the model looks like. The comparative experiments, instead, modify the same parameter multiple times to see how different models behave in order to choose the solution that is the closest to the real solution or that better optimizes the model parameters.

As already dealt, the starting point in a simulation project is the identification of a problematic reality that will be analyzed in order to highlight inside the system to be studied. The system is not reality, but it is a representation, where the detail level will depend on the objectives of the study and the type of problem that must be resolved. Then the objectives are defined and we can proceed with the construction of a formal model that allows to simulate the identified system, in order to understand the behavior. The process outlined in this way, starts from reality up to the final decisions in three phases: the identification of the system that must be studied, modeling and simulation.

In a simulation model at any given time the system is described by a set of variables called "state variables". In case of variables vary continuously, we consider a continuous system, however in case of variables change instantaneously at precise instants of time, belonging to a finite or countable set, we refer to a discrete system.

Considering this aspect it is important to observe that the choice of a model with respect to another is not strictly linked to the type of the system but to the objectives to be pursued and therefore the type of study to be performed. Once the model is built, it needs to implement the simulation model using a *general purpose* language like (i.e.: Java, C, Smalltalk and so on). For this purpose there are libraries of 'routines' oriented to the simulation or specialized languages typical for the simulation such as, for example, among the best known SIMSCRIPT, ModSim, that provide many features typical for the realization of a model with the advantage of reducing the manufacturing time.

Another possible alternative, to take into consideration, are commercial or open source software applications already created ad hoc for the simulation, the so-called simulators such as Simul8, Simulation Arena, Extend, or Vensim among the many available. It is a software packages interactive type and equipped, in general with a graphical interface, then very intuitive and easy to learn, however, because they are dedicated to specific types of systems, they have the disadvantages of being limited to specific model systems[71]. Here are some of the advantages to using modeling and simulation:

- the accuracy and ability to choose correctly by testing every aspect of a proposed change without committing additional resources;

- compress and expand time to allow the user to speed up or slow-down behavior or phenomena to facilitate an in-depth research;
- understand all aspects of the scenario in order to better understand and reproduce it;
- explore possibilities in the context of policies, operating procedures, methods without disrupting the actual or real system
- diagnose problems by understanding the interaction among variables that make up complex systems;
- identify constraints by reviewing delays on process, information, materials to ascertain whether or not the constraint is the effect or the cause;
- develop understanding by observing how a system operates rather than predictions about how it will operate;
- visualize the plan with the use of animation to observe the system or organization actually operating;
- build consensus for an objective opinion because Software Process Simulation can avoid inferences;
- prepare for change by answering the 'what if' in the design or modification of the system;
- invest wisely because a simulation study costs much less than the cost of changing or modifying a system;
- better training can be done in a less expensive way and with less disruption than on-the-job training;
- specify requirements for a system design that can be modified to reach the desired goal.

It is obvious that there are many uses and many advantages to using Software Process Simulation. The IIE also made note of some disadvantages to using Software Process Simulation. The list is noticeably shorter and it includes things such as:

- the special training needed for building models;
- the difficulty in interpreting results when the observation may be the result of system inter-relationships or randomness;
- cost in money and time due to the fact that simulation modeling and analysis can be time consuming and expensive;
- inappropriate use of modeling and simulation when an analytical solution is best.

4.3 Common uses of simulation modelling

The main purposes of simulation models, as proven by Raffo et al. [4], are the follwing: to provide a basis for experimentation, predict behavior, answer "what if" questions, teach about the system being modeled, etc.

A software process simulation model faced some particular aspects of software development/ maintenance/evolution process. It can represent such a process as currently implemented (as-is), or as planned for future implementation (to-be). Since all mo dels are abstractions, a model represents only some of the many aspects of a software process that potentially could be modeled namely the ones believed by the model developer to be especially relevant to the issues and questions the model is used to address. There is a wide variety of reasons for undertaking simulations of software process models. In many cases, simulation is an aid to decision making. It also helps in risk reduction, and helps management at the strategic, tactical, and operational levels. Here we have a list of the main reasons for using simulations of software processes.

- *Strategic management.*

  Simulation can help to address a broad range of strategic management questions regarding work distribution across sites or centralization at one location and or if it is better to perform work in-house or to out-source. Likewise analysis of benefits regarding the employment of a product-line approach to developing similar systems, or a more traditional approach, or by proposing individual product development approach.

  In each of these cases, simulation models would contain local organizational parameters and be developed to investigate specific questions. Managers are assisted in their decision making by comparing the results that come from simulation models of the alternative scenarios.

- *Planning.* Simulation can support management planning in a number of clear ways, including

  * forecast effort / cost, schedule, and product quality;
  * forecast staffing levels needed across time;
  * cope with resource constraints and resource allocation;
  * forecast service-level provided (e.g., for product support);
  * analyze risks.

  All of these can be applied to both initial planning and subsequent re-planning. Simulation can also be used to help select, customize, and accomplish the best process for a specific project context. These are process planning issues.

- *Control and operational management.*

  Simulation can also provide effective support for managerial control and operational management. Simulation can facilitate project tracking and oversight because key project parameters (e.g., actual status and progress on the work products, resource consumption to-date, and so forth) can be monitored and compared against planned values computed by the simulation. This helps project managers to determine, when it is possible, if corrective action may be needed. Project managers can also use simulation to support operational decisions, such as whether to commence major activities (e.g., coding, integration testing). To do this, managers would evaluate current project status using current actual project data and employ simulation to predict the possible outcome if a proposed action (e.g., commence integration testing) taken then or delayed.

– *Process improvement and technology adoption.*

Simulation can support process improvement and technology adoption in a variety of ways. In process improvement settings, organizations are often faced with many suggested improvements. Simulation can aid specific process improvement decisions (such as go / no-go on any specific proposal, or prioritization of multiple proposals) by forecasting the impact of a potential process change before putting it into actual practice in the organization.

These applications, use simulation a priori to compare process alternatives, by comparing the projected outcomes of importance to decision makers (often cost, cycle-time, and quality) resulting from simulation models of alternative processes.

Simulation can also be used ex post to evaluate the results of process, changes or selections already implemented. The actual results observed would be compared against simulations of the processes not selected, after updating those simulations to reflect actual project characteristics seen (e.g., size, resource constraints). The actual results would also be used to calibrate the model of the process that was used, in order to enhance future use of that model.

Just as organizations face many process improvement questions and decisions, the same is true for technology adoption. The analysis of inserting new technologies into a software development process (or business process) would follow the same approach as for process change and employ the same basic model. This is largely because adoption of a new technology is generally expected to affect things that are usually reflected as input parameters to a simulation (e.g., defect injection rate, coding productivity rate) and / or to change the associated process in other more fundamental ways. about software processes in several ways.

– *Understanding.*

Simulation can promote enhanced understanding of many process issues. For example, simulation models can help project managers, software developers, and quality assurance personnel better understand process flow, i.e., sequencing, parallelism, work and products flows, etc. Simulation results could help people visualize these process flow issues and could be presented using Animated simulations, Gantt charts and so on. And, also, to understand the effects of the complex feedback loops and delays inherent in software processes;

In addition, simulation models can help researchers to identify and understand consistent, pervasive properties of software development and maintenance processes. Moreover, simulations (especially Monte Carlo techniques) can help people understand the inherent uncertainty in forecasting software process outcomes, and the likely variability in actual results seen. Finally, simulations could facilitate communication, common understanding, and consensus building within a team or larger organization and likewise help with process or organizational understanding to some degree.

– *Training and learning* .

Simulation can help with training and learning about software processes in different ways. Although this purpose is closely related to that of understanding, the particular setting envisioned here is an explicitly instructional one. Simulations can provide an useful way for personnel to practice/learn project management.

> A simulated environment can help management trainees learn the likely impacts of common decisions (often mistakes), e.g., rushing into coding, skipping inspections, or reducing testing time. Finally, training through simulation can help people to accept the unreliability of their initial expectations about the results of given actions; most people do not possess good skills or inherent abilities to predict the behavior of systems with complex feedback loops and/or uncertainties (as are present in software processes).

Software process simulation modeling has gained increasing interest among academic researchers and practitioners as an useful approach for analyzing complex business.

Simulation modeling has been applied in a variety of disciplines for a number of years, it has only recently been applied to the area of software development and evolution processes.

The scope of software process simulation applications ranges from the strategic management of software development, supporting process improvements,to software project management training and longer-term product evolutionary models, in general.

A software process has been specifically defined as *A set of activities, methods, practices, and transformations used by people to develop and maintain software products*. Likewise, model is an abstraction (i.e., a simplified representation) of a real or conceptual complex system. A model is an useful tool to explain the main features and characteristics of the system, all oriented to study, predict, modify, or control it.

In general, the model could include some, but not all, aspects of the system being mo deled. A model is valuable to the extent that it provides useful answers to the questions it is used to address. It could represent dynamic system.

One of the main purposes for developing a simulation model is to gain important insights when the costs (in time and resources) and risks are prohibitive. Simulations are generally employed when the complexity of the system being modeled is beyond what static models or other techniques can usefully represent.

4.4 Simulation techniques and approaches

When we decide to design and to implement a simulator, various techniques and strategies might be adopted to model the behavior of a given system.

According to the characteristics of the system to be simulated, some techniques may be more suitable than others. The main factors to take into consideration are the level of abstraction and the desired accuracy and speed of the simulation. Traditionally, simulators are designed using either continuous or discrete-event techniques to simulate a given system [19].

Also, it is useful to classify the system being simulated into two separate categories depending upon the degree of randomness associated with the behavior of the system in its simulated environment. As such, the simulation results for a given set of inputs will always be identical [19].

4.4.1 Continuous Simulation

A research group at the Massachusetts Institute of Technology (MIT) developed a societal model of the world. This was the first continuous simulation model: the World Model. Today, most continuous models are based on differential equations and/or iterations, which use several input variables for calculation and in turn supply output variables. The model itself consists of nodes connected through variables. The nodes may be instantaneous or noninstantaneous functions. Instantaneous functions present their output at the same time the input is available. Noninstantaneous functions, also called memory functions, take some time for their output to change. Even complex functions containing partial differential equations that would be difficult or impossible to solve analytically or numerically may be modeled using the three basic components mentioned above. Before computers became as powerful as they are today, the analog approach was the only way to solve this kind of equations within a reasonable amount of time. Due to the continuous nature of the *solver*, the result could be measured instantly.

In software engineering contexts, continuous simulation is used primarily for large-scale views of processes, like the management of a complete development project or strategic company management. Dynamic modeling enables us to model feedback loops, which are very numerous and complex in software projects. Of course, simulating this continuous system on a computer is not possible due to the digital technology used. To cope with this, the state of the system is computed at very short intervals, thereby forming a sufficiently correct illusion of continuity. This iterative recalculating makes continuous models simulated on digital systems grow complex very fast.

With continuous simulation, time is controlled by continuous variable expressed as differential equations. During the simulation of the software the equations will be integrated. The more popular approach for simulating in a time-continuous way is the System Dynamics modeling [33]. Abdel-Hamid was the first person to use system dynamics for modeling software project management process [44] System dynamics models describe the system in terms of *flows* that accumulate in various levels. The flows can be dynamic functions or can be the consequence of other *auxiliary* variables. As the simulation advances time in small evenly spaced increments, it computes the changes in levels and flow rates . For example, the error generation rate may be treated as a *flow* and the current number of errors could be treated as a *level*. Because system dynamics models deal with flows and levels, there are no individual entities and thus no entity attributes. For a software process model, this means that all modules and all developers are equal. If we wanted to model the effect of a few error prone code units on the development process, we would not be able to specify which units were error-prone [24]. In system dynamics, a system is defined as a collection of elements that continually interact with each other and outside elements over time, as a whole system [35]. The two important elements of the system are structure and behavior. The structure is defined as the collection of components of a system, and their relationships. The structure of the system also includes the variables that are important in influencing the system. The behavior is defined as the way in which the elements or variables composing a system vary over time [35]. While it is possible to represent discrete activities in a system dynamics model, the nature of the tool implies that all

levels change at every time interval [24]. If the process contains sequential activities, some mechanism must be added to prevent all activities from executing at once. For example, if we modeled the software process as define, design, code and test activities, as soon as some code was defined, design would start. If we wanted to model a process that completed all design work before coding started, we would have to create an explicit mechanism to control the sequencing [24]. While the system dynamics model is an excellent way to describe the behavior of project variables, it is a more difficult way to describe process steps. Finally, a system dynamics model does not easily allow to model the uncertainty inherent in our estimates of model parameters. A discrete model could sample from a distribution using a different time for each module. The system dynamics model either must sample the rate at each time step or must use the same rate for each model run [24].

4.4.2 Discrete Event Simulation

The discrete approach shows parallels to clocked operations like those used by car manufacturers in production. The basic assumption is that the modeled system changes its state only at discrete moments of time, as opposed to the continuous model. So, every discrete state of the model is characterized by a vector containing all variables, and each step corresponds to a change in the vector. Let us consider a production line at a car manufacturer. Simplified, there are parts going in on one side and cars coming out on the other side. The production itself is clocked: Each work unit has to be completed within a certain amount of time. When that time is over, the car-to-be is moved to the next position, where another work unit is applied. (In reality, the work objects move constantly at a very low speed. This is done for commodity reasons and to realize minimal time buffer. Logically, it is a clocked sequence.) This way, the car moves through the complete factory in discrete steps. Simulating this behavior is easy with the discrete approach. Each time a move is completed, a snapshot is taken of all production units. In this snapshot, the state of all work units and products (cars) is recorded. At the next snapshot, all cars have moved to the next position. The real time that passes between two snapshots or simulation steps can be arbitrary. Usually the next snapshot of all variables is calculated and then the simulation assigns the respective values. Since the time needed in the factory to complete a production step is known, the model appropri- ately describes reality. A finer time grid is certainly possible: Instead of viewing every clock step as one simulation step, arrival at and departure from a work position can be used, thereby capturing work and transport time independently. The discrete approach is used in software engineering as well. One important area is experimental software engineering, e.g., regarding inspection processes. Here, a discrete simulation can be used to describe the process flow. Possible simulation steps might be the start and completion of activities and lags, together with special events like (late) design changes. This enables discrete models to represent queues.

Discrete event simulation [36] involves modeling a system as it progresses through time and is particularly useful for analyzing queuing systems. Such systems are common in the manufacturing environment and are obvious as work in progress, buffer stocks, and warehouse parts.

A major strength of discrete event simulation is its ability to model random events and to predict the effects of the complex interactions between these events. Experimentation is normally carried out using the software model to answer "what-if?" questions. This is achieved by changing inputs to the model and then comparing the outcomes. This type of simulation is primarily a decision support tool. Inside the software or model will be a number of important concepts, namely entities and logic statements. Entities are the tangible elements found in the real world, e.g. for manufacturing these could be machines or trucks. The entities may be either temporary (e.g. parts that pass through the model) or permanent (e.g. machines that remain in the model). The concepts of temporary and permanent are useful aids to understanding the overall objectives of using simulation, usually to observe the behavior of the temporary entities passing through the permanent ones. Logical relationships link the different entities together, e.g. that a ma- chine entity will process a part entity. The logical relationships are the key part of the simulation model; they define the overall behavior of the model.

Each logical statement (e.g.*start machine if parts are waiting*) is simple but the quantity and variety and the fact that they are widely dispersed throughout the model give rise to the complexity. Another key part of any simulation system is the simulation executive. The executive is responsible for controlling the time advance. A central clock is used to keep track of time. The executive will control the logical relationships between the entities and advance the clock to the new time. The process is illustrated in Figure 2.1. The simulation executive is central to providing the dynamic, time based behavior of the model. Whilst the clock and executive are key parts of a simulation system they are very easy to implement and are extremely simple in behavior.Two other elements that are vital to any simulation system are the random number generators and the results collation and analysis. The random number generators are used to provide stochastic behavior typical of the real world.

The model is advanced to the time of the next significant event. Hence if nothing is going to happen for the next 3 minutes the executive will move the model forward 3 minutes in one go. The nature of the jumping between significant points in time means that in most cases the next event mechanism is more efficient and allows models to be evaluated more quickly. The event approach is described in Figure 4.1. The diagram shows two essential elements: the clock and simulation executive. Here the simulation executive will use an event list (a string of chronologically ordered events). The executive is responsible for ordering the events. The executive removes the first event from the list and executes the relevant model logic. Any new events that occur as a result are inserted on the list at the appropriate point (e.g. a machine start load event would generate a machine end load event scheduled for several seconds time). The cycle is then repeated.

Each event on the event list has two key data items. The first item is the time of the event which allows it to be ordered on the event list. The second item is the reference to the model logic that needs to be executed. This allows the executive to execute the correct logic at the correct time. Note that more than one event may reference the same model logic, this means that the same logic is used many times during the life of the simulation run [28], [25]. Discrete event simulation is efficient and particularly appealing when the process is viewed as a sequence of activities, such as in a manufacturing line where items or entities move from station to station and have processing done at

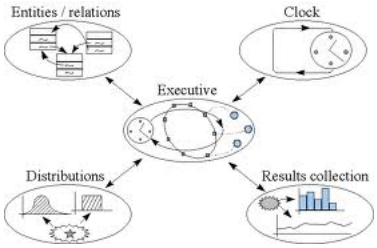

Figure 4.1: *Structure of Discrete Events System.*

each station. Discrete models easily represent queues and can delay processing at an activity if resources are not available. In addition, because discrete models are based on the idea of a sequence of activities, it may be awkward to represent simultaneous activities. While activities can occur in parallel, it is difficult to represent the idea of an entity in two activities simultaneously. Imagine a code module in which some parts are in coding while other parts are in unit test. To capture this in a discrete model, we are forced to model sub-components of the module so that each sub-component can be in only one activity at a time [24].

4.4.3 Agent-Based Simulation

A simulation model based on agents (the so-called Agent-based Simulation, henceforth ABS) is a multi-agent system consisting of multiple intelligent agents interaction.

An agent is a computational entity that has the ability to carry out independent actions in a typical dynamic and non-deterministicenvironment. The multiagent systems have been developed primarily with the aim of solving complex problems. So the agent represents the basic unit during the simulation interacts with other agents, creating structures articulated by complex evolutionary dynamics.

An agent has certain characteristics such as: Autonomy: Agents are at least partially independent and able to interact with each other; Location: No agent has a completely full vision of the system, but it may have a local knowledge of what surrounds him; Decentralization: No type of hierarchy exists among the agents of a system (no agent plays the role of controller); Adaptability: Agents are responsible for maintaining their own state. In an ABS system the overall behavior emerges from the independent interaction of agents see in fig 4.2.

ABS models are now emerging as a tool to solve problems related to complex systems of interacting entities. Unique characteristics of the ABS model are the ability to perform simulations in the long term and large-scale, and the combination of the software process (like open source development), processes and agile software development.

4.4.4 Hybrid Simulation

Currently, research is focusing on the analysis and the study of such hybrid simulation models, i.e those models generated by the integration of two or more above men-

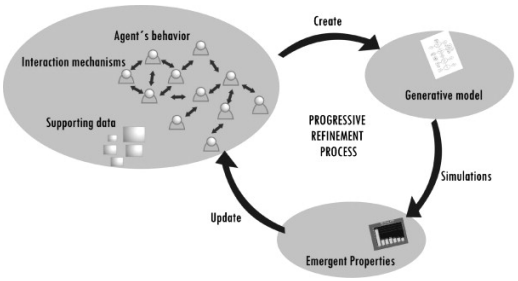

Figure 4.2: *Structure of Agent based System.*

tioned simulation models. According to a study by J. He Zhang and other researchers [5], hybrid models are now capturing the interest in the field of research increasingly relevant, as they would provide the ability to capture more realistically the complexity of the real world. In fact, the simulation models hybrids are born with the aim of being able to solve in a single model problems with enough complexity and amplitude that no simulation approach taken individually, such as SD, SED, or ABS, for example, might do.

The name "hybrid simulation" is a contraction of computational capabilities analog (continuous part) and digital (discrete part) used together to define the model.

For this reason it is possible to use this model in order to study the problems of processes that affect some other continuous and discrete variables. When you want to then deal with the process simulation software using a hybrid model, as claimed by Kellner, Madachy and Raffo [4], it is necessary to first find an answer to the following question: 'What aspects of this particular software process it is better to represent as continuous and such others as discrete?'

If the system under analysis is very simple, it is also easy to determine and separate these aspects. But if the system is very complex, to find an answer to this question can avail itself of a number of software tools that allow you to determine both the aspects continuous and discrete ones although it is very difficult to determine a clear distinction between the two parties, see fig 4.3.

For this reason, the research is moving on this front, in such way as to achieve a good identification of what is continuous and of what is rather discrete.

Specifically, the software process shows both the appearance of a discrete system than that of a continuous system, i.e. is characterized by dynamic events that evolve in a dynamic time. Since the discrete-event model describes the development of the software process as a sequence of discrete activities, namely the change in the value of state variables occurs at discrete moments, this model can not therefore have enough events to represent the continuous variable dynamic process.

On the other hand the dynamic models describe the interactions between the design factors, but they are not easily separate process steps. In the literature there are numerous attempts to combine the simulation of discrete event simulation of continuous events to model more realistically the software processes. For example Rus et al [30] and Lakey [31] have shown the process as a discrete activities and have tried to incorporate the mechanism of feedback cycle typical of a dynamic system. Martin and Raffo

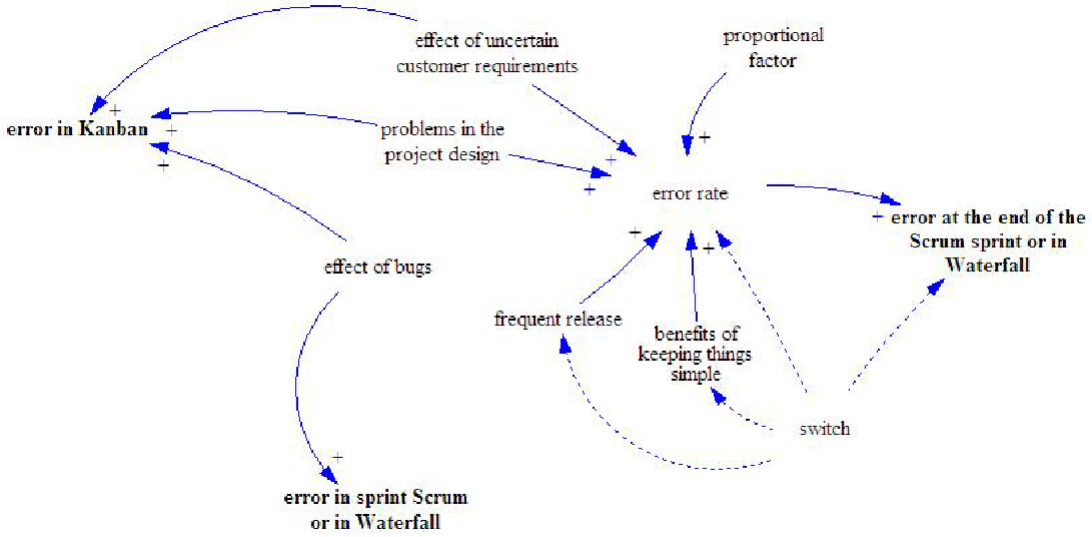

Figure 4.3: *Example of the use System Dynamics.*

[4] [29] analyzed the use of labor by means of a hybrid simulation model. Donzelli and Iazeolla [22] have finally combined the three traditional models modeling (analytical, continuous and discrete event).

Continuous simulation models describe the interaction between project factors well, but suffer from a lack of detail when it comes to representing discrete system steps. Discrete simulation models perform well in the case of discrete system steps, but make it difficult to describe feedback loops in the system. To overcome the disadvantages of the two approaches, a hybrid approach as described by Martin and Raffo [24] can be used. In a software development project, information about available and used manpower is very important.

While a continuous model can show how manpower usage levels vary over time, a discrete model can point out bottlenecks, such as inspections. Because of insufficient manpower, documents are not inspected near-time, so consumers of those documents have to wait idly, which wastes manpower.

In a purely discrete approach, the number of inspection steps might be decreased to speed up the inspection process. While possibly eliminating the bottleneck, this might introduce more defects. The discrete model would not notice this until late in the project because continually changing numbers are not supported.

The hybrid approach, however, would instantly notice the increase, and depending on how the model is used for steering the project more time would be allocated for rework, possibly to the extent that the savings in inspection are overcompensated. Thus, the hybrid approach helps in simulating the consequences of decisions more accurately than each of the two single approaches does individually.

Chapter 5

Model Description

5.1 Simulation Model Description

The Kanban system is a new approach to controlling variability in software development processes, and to catalyzing process improvements and a culture of continuous improvement.

The Kanban concepts are simple, structured and yet powerful. For these reasons, it is possible to design a software process simulator that captures most of what happens during development. We developed an event-driven Kanban simulator using a full object-oriented approach, in Smalltalk language.

In the simulator, software requirements are decomposed in features, or work items, that can be independently implemented. The implementation is accomplished through a continuous flow across different activities, from requirement analysis to deployment. The work is performed by a team of developers, able to work on one or more activities, depending on their skills.

We devised a paradigmatic setting of a typical project, and used the simulator to assess the optimum maximum number of features that can be worked in an activity, at any given time – the Kanban limits.

We developed an event-driven simulator of the Kanban software process – a WIP limited pull system visualized by the Kanban board – and used this simulator to assess the optimum values of the working item limits in the activities. Initially we studied a paradigmatic case of 4 activities and a given number of work items, defining a cost function equal to the total time needed to complete the project, plus a weighted sum of the limits themselves. We performed an exhaustive search on all the admissible values of the solution, finding sensible optimal values, and a non-trivial behavior of the cost function in the optimization space. This demonstrates the feasibility and usefulness of the approach.

To be able to simulate a Kanban software development process, we devised a model of system described that is presented in this section. This model is simple enough to be manageable, but is able to capture all relevant aspects of the process. Moreover,

the model itself and its object-oriented implementation allow to easily adding further details, if needed.

The simulation model records all the significant events related to the features, developers and activities during the simulation, to be able to compute any sort of statistics, and to draw diagrams, such as the cumulative flow diagram. In the next section we recall each actor of the mentioned model describing the main aspects and its relationships.

5.2 Description of the Actors

This chapter is dedicated to describe the basic simulation model and its entities. For each actor will be given the relationship with the others, showing the equation that guide the inputs of the simulation. In the next section a customization, in order to reproduces each case study, is done.

5.2.1 Developers

The present simulator allows for a single team working on the features. The team is composed of N_D. developers. The typical index for a developer is denoted by j. Each developer is characterized by a list of her skills, one for each activity, $s_{j,k}, k = 1,...NA$. A skill is characterized by the following parameters:

- **minValue** ($min_{j,k}$): the minimum possible value of the skill in $k-th$ activity for $j-th$ developer, when her/his experience is zero.
- **maxValue** ($max_{j,k}$): the maximum possible value of the skill in $k-th$ activity for $j-th$ developer, when her/his experience is maximum.
- **experience** ($ex_{j,k}$): the number of days the $j-th$ developer has been working on feature development in $k-th$ activity.
- **maxExperience** ($maxEx_{j,k}$): the number of working days needed to reach the maximum experience of $j-th$ developer in $k-th$ activity, at which point the skill reaches its maximum value, and does not increase anymore.

The actual value, $v_{j,k}$, of skill $s_{j,k}$ for $k-th$ activity, where $j-th$ developer worked for $ex_{j,k}$ days is given by a linear interpolation, shown in eq. 5.1.

$$v_{j,k} = min_{j,k} + \frac{(max_{j,k} - min_{j,k})ex_{j,k}}{maxEx_{j,k}},\ if\ ex_{j,k} \leq maxEx_{j,k} \qquad (5.1)$$
$$max_{j,k}\ if\ ex_{j,k} > maxEx_{j,k}$$

Clearly, the use of another form of interpolation, such as for instance a logistic, would be easy. If we wish to simulate a system where no learning takes place, we can set $min_{j,k} = max_{j,k}$, with $maxEx_{j,k} > 0$. The value of $ex_{j,k}$ will in any case record the days $j-th$ developer has been working on $k-th$ activity. If the skill factor $s_{j,k}$ is equal

to one (the default case), the time spent by the $j-th$ developer on a feature exactly matches the amount of effort dedicated to it. If $s_{j,k} > 1$, the developer is more clever than average, and her time spent working on the feature has a yield greater than the actual time. On the contrary, If $s_{j,k} < 1$, it will take a time longer that the actual effort to perform the work on the feature.

Each developer works on a feature (in a specific activity) until the end of the day, or until the feature is completed. When the state of the system changes, for instance because a new feature is introduced, a feature is pulled to an activity, work on a feature ends, and in any case at the beginning of a new day, the system looks for idle developers, and tries to assign them to features available to be worked on in the activities they belong to. For each idle developer, the following steps are performed:

1. the activities s/he is most skilled in are searched in decreasing order of skill;
2. for each activity, the features waiting to be worked on are considered, in decreasing order of relevance;
3. the first feature in the list found, if any, is assigned to the developer; let it be feature $i-th$ in activity $k-th$;the time $t'_{i,k}$to finish the feature is computed, taking into account the total estimated effort, $y_{i,k}$, the work already performed on the feature, $w_{i,k}$, the developer's skill in the activity, $s_{j,k}$, and a further penalty, p, applied if the developer was not already working on the feature the day before. The time to finish the feature is:

$$t'_{i,k} = p\frac{y_{i,k}w_{i,k}}{s_{j,k}} \tag{5.2}$$

The penalty factor p is equal to one *(no penalty)* if the same developer, at the beginning of a day, works on the same feature s/he worked the day before. If the developer starts a new feature, or changes feature 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 feature. In this case, the value of p is greater than one. Let us call T the current simulation time, and T_e the time marking the end of the working day.

1 . If $T + t'_{i,k} > T_e$, the work on the feature is limited until the end of the day, so it is performed for a time $t_e = T_e - T$. The actual implementation effort considered, $w'_{i,k}$, is proportional to t_e, but also to the skill, and inversely proportional to the penalty factor p:

$$w'_{i,k} = \frac{t_e s_{j,k}}{p} \tag{5.3}$$

The new work performed on the feature becomes: $w_{i,k}(T_e) = w_{i,k}(T) + w'_{i,k}$.

2 .If $T + t'_{i,k} \leq T_e$, the work on the feature in k-th activity ends within the current day, the feature is moved to the *"Done"* state within the activity, and the developer is ready to start working on another feature.

If the skill of a developer in an activity is below a given threshold (set to 0.4 in the current simulator) s/he will never work in that activity(see fig. 5.1).

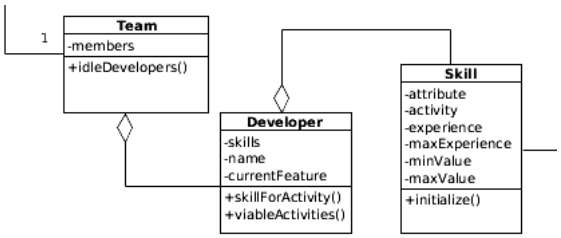

Figure 5.1: *UML Class Diagram of Developer with their skills, and organized in Team.*

5.2.2 Features and Activities

Features

The project is decomposed into atomic units of work, that are not further decomposed. They are considered as already specified in a phase preceding the work simulated, and are given as inputs to the simulator.

New features can be added as time proceeds. In the followings, the typical index for a feature is denoted by i, while the total number of features at time t is $N_F(t)$. Each feature is characterized by a name, a state and an estimate e_i â expressing how long it will take to implement the $i-th$ feature, in man-days. The features assigned to an activity can be in three different states:

- *just pulled but not yet assigned*;
- *under work*;
- *done, but not yet pulled by the next activity*.

Other possible states are *Backlog* and *Released*, denoting the initial or final state, respectively. In the present model, the features are just atomic, and are not explicitly linked to a specific project (see fig. 5.2). It would be possible, however, to introduce different projects, and assign the features to them. The activities represent the work to be done on the features. They can be freely configured.

Activities

In the followings, the typical index for an activity is denoted by k, while the total number of activities is N_A. Each activity is characterized by:

- **name**: the name of the activity.
- **maxFeatures**: the maximum number of features that can be handled at any given time, including features in every possible state. It is denoted by $M_K$for the $k-th$ activity.
- **percentage**: the typical percentage of the total estimated cost of a feature that pertains to the activity. For instance, is a feature has an overall estimate of 10 days, and the Design activity has a percentage of 15, the design of the feature will

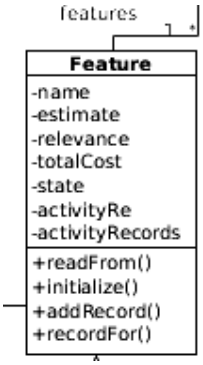

Figure 5.2: *UML Class Diagram of Features*

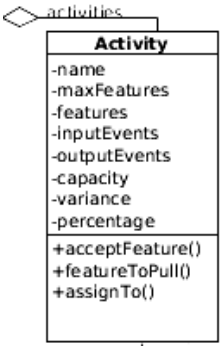

Figure 5.3: *UML Class Diagram of Activities*

be estimated to be 1.5 days. The sum of the percentages of all the activities should be one. It is denoted by pk for the $k-th$ activity, with the constraint:

$$\sum p_k = 1. \tag{5.4}$$

When work starts on feature $i-th$ within a given $k-th$ activity, the actual effort of development of the feature in the activity is computed. If the initial total estimate of the feature is e_i, and the percentage of the total cost due to the activity is p_k, the starting estimate of the work is $x_{i,k} = e_i p_k$. The estimate $x_{i,k}$ can be randomly perturbed, by increasing or decreasing it of a percentage drawn from a given distribution D â in such a way that the average of a high number of perturbations is equal to the initial estimate $x_{i,k}$. A way to obtain this behavior is to multiply $x_{i,k}$ by a random number, r, drawn from a log-normal distribution with mean value equal to 1: $y_{i,k} = x_{i,k}$, where yi,k is the actual value of the effort needed to work on feature $i-th$ in activity $k-th$. If this feature is assigned to developer $j-th$, the time to perform the work depends also on the developer's skill in activity $k-th$, $s_{j,k}$. The skill is a parameter denoting the developer's ability with respect to an "average" developer in a given activity. The estimate of the effort reflects the time to complete a task by the "average" developer. So this time, t_{ik}, expressed in days to complete an activity whose estimate is $y_{i,k}$, in function of the skill is: $t_{i,k} = \frac{y_{i,k}}{s_{j,k}}$. So, if the skill is equal to one the developer acts as an "average" developer and the time is equal to an estimate, if the skill is greater(smaller)than one the time is lower(longer). In the current implementation of the simulator, only a single developer can perform an activity on a feature in a given time (no pair-programming). Since the cost of all developers is considered equal, the actual time, t_{ik}, is directly proportional to the cost of implementing the feature. The total cost of implementing a feature is thus proportional to the sum of the development times in the various activities, t_i:

$$t_i = \sum_{k=1}^{N_A} t_{i,k}. \tag{5.5}$$

The sequence of activities the implementation of each feature proceeds through is

fixed for each simulation. An example of activities might be: *Design*, *Coding*, *Testing*, *Integration*, but what really matters in the simulation is their number and their operational characteristics (see fig. 5.3).

5.2.3 Events

The simulator is event-driven, meaning that the simulation proceeds by executing events, in order of their time. When an event is executed, the time of the simulation is set to the time of the event. The simulator holds an event queue, where events to be executed are stored sorted by time, and which the events are taken from to be executed(see fig 5.4). When an event is executed, it changes the state of the system, and can create new events, with times equal to, o greater than, the current time, inserting them into the event queue. The simulation ends when the event queue is empty, or if a maximum time is reached. The time 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 week-ends, holidays, or overtime. A day can be considered to have 8 nominal working hours, or less if the time lost on average for organization tasks is accounted for. If we want to consider calendar days, it is always possible to convert from nominal working days to them. For instance, let us suppose that the starting day of the project is Monday, 25 October 2010, and that the current time is 7.5 days. Since 30 and 31 October are a week-end, and November 1st is holiday, the 7th day after the start is November 3rd, and the 8th day is November 4th. So, day 7.5 happens in the middle of November 4th, say at noon if the working day starts at 8 a.m. and ends at 17 p.m., with a lunch interval between noon and 1 p.m.

The relevant events of the simulation are the followings:

- **FeatureCreation**: a new issue (the so called feature) is created and inserted into the Backlog, and an event **FeatureToPull** is generated for the first activity, at the same time. This event refers only to features introduced after the start of the simulation, and not to the features initially included in the Backlog.
- **FeatureWorkEnded**: the work of a developer on a feature, within a given activity, has ended, and the developer becomes idle. This may happen when the feature is actually finished, as regards the activity, or at the end of the working day. In the former case, the state of the feature is changed, and an event FeatureToPull is generated for the next activity, at the same time.
- **FeatureToPull**: an activity is requested to pull a feature from the previous activity – or from the Backlog if it is the first. If the activity is nil, it means that a feature in the state of *Finished* should be moved from the last activity to the *Released* state. If the activity has already reached its maximum number of features, or if there is no feature to pull in the previous activity, nothing happens. If there is a feature that can be pulled, it is pulled to the activity, and another event **FeatureToPull** is generated for the previous activity (if the activity is not the first), at the same time. All idle developers of the team are asked to find a feature ready to be worked. If a developer finds such a feature, the developer is assigned to the feature, the actual time taken to complete the work is computed (according to eq 5.2), and a FeatureWorkEnded 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 feature is finished).

The three events are enough to manage the whole simulation. The simulation is started by creating the starting features, putting them in the Backlog, generating as many **FeatureToPull** events (at $time = 0$) for the first activity as its **maxFeatures** value, and then generating as many **FeatureCreation** events for future times as required. The simulator is then asked to run, using the event queue created in this way. When the work on all features has been completed in all activities, no more feature can be pulled and no more work can start, so the event queue becomes empty, and the simulation ends. To represent different kind of process, some other event was added like i.e. *StartIteration* typical of Scrum process. This event 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.

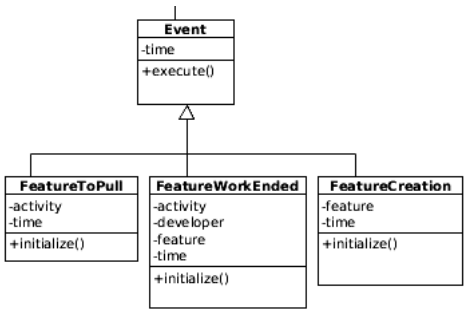

Figure 5.4: *UML Class Diagram of Events.*

5.3 The Object-oriented Model

In Fig 5.5 we present the UML class diagram of the OO model of the simulator, showing the classes of the system, and their relationships. The high-level classes of the model are the simulator itself, the KanbanSystem – which includes the software project and the process – the team of developers. Lower-level classes are Activity,Feature, Developer, Skill, and three kinds of Events. Utility classes used to record data for further analysis are WorkRecord, ActivityRecord and FeatureChanged.

In this class diagram there are the different actors of simulation, *who* works-*the developer*-*what* is developed-*the feature*- *when* the work is performed-*the events*-and finally *what* is performed- *the activities*. The simulator is implemented in Smalltalk language, a language very suited to event-driven simulation, and very flexible to accommodate any kind of changes and upgrades to the model. We design a simulator to simulate the software maintenance process. Our simulator is event-driven, meaning that the simulation proceeds by executing events, in order of time. When an event is executed, the current time of the simulation is advanced to the time of the event.

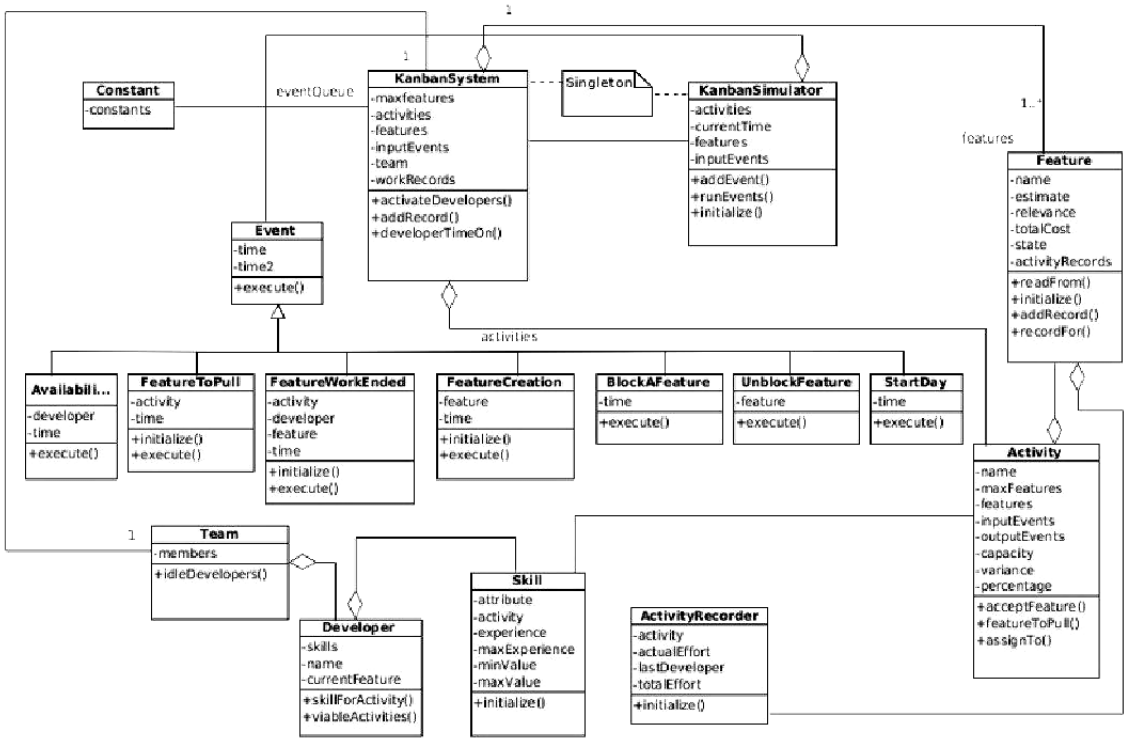

Figure 5.5: *UML Class Diagram of Simulation Model.*

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 8 nominal working hours.

A particular mention of KanbanSystem and KanbanSimulator classes (see fig. 5.6) that represent the engine of the whole simulator. These classse are responsible to "run" the simulation through some scripts that recall the suitable methods regarding events, features, activities and finally the plot of the output like statistic of cycle and lean times or CFD(cumulative flow Diagram) in order to calculate WIP, Throughput and totalTime required.

5.4 Calibration and Validation

The simulation model, presented in this Chapter, has been calibrated and validate according with different case studies in the Chapter 6 "Applications of the Simulation

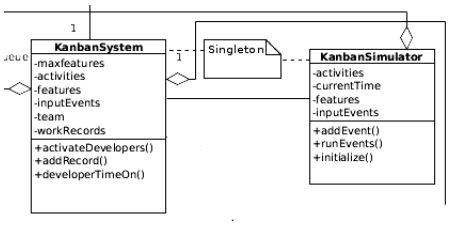

Figure 5.6: *UML Class Diagram of KanbanSystem and KanbanSimulator.*

Model". For each process the model was modified in order to follow and reproduce the used process. At the end of each case study a comparison among results that come from the project and simulate case is done. This is in order to demostrate the reliability of the simulation model and to show the repeatibility of the input parameters and the results.

Chapter 6

Applications of the Simulation Model

In this section we presented the case studies analyzed and simulated in order to show the abilities of the simulation model when is applied to real cases. We started presenting the first instance of the simulation model regarding a software development process and the optimization of its limits. After, we presented the case studies regarding a software maintenance process and its model and reproduction using an original process (TSP/PSP), a Kanban process and a Scrum process. Then, the case regarding a maintenance process using data and process collected by a Chinese firm and Microsoft case. We performed a comparison between original processes and WIP-limited in order to show the improvements achieved through the us of a WIP-Limited approach And in the case of risk management we performed a comparison between two different approaches Scrum and Lean-Kanban.

We offered a description of each case study organized as follows. First we presented the description of the case study, then a description of the applied approach and in the last section the calibration of the model in order to follow the used process.

6.1 Case Study One

The main goal of this part of our research was to better understand the Lean-Kanban approach, to evaluate its effectiveness, and to develop methodologies to optimize its parameters.

To this purpose, we developed an event-driven, object-oriented simulator of Kanban systems, both to analyze them using the simulation approach, and to optimize their parameters. Using simulation optimization, the practice of linking an optimization method with a simulation model to determine appropriate settings of certain input parameters is possible to maximize the performance of the simulated system [14].

6.1.1 Description of the approach

In general, we can define the Kanban software process as a WIP limited pull system visualized by the Kanban board. Recently, the Kanban approach applied to software development, seem to be one of the hottest topics of Lean. In the recent 3-4 years, Kanban has been applied to software process, and is becoming the key Lean practice in this field.

A correct use of the Kanban board helps to minimize WIP, to highlight the constraints, and to coordinate the team work. However, Lean is more than Kanban, and more Lean practices should be used, together with Kanban, to take full advantage of the application of Lean to software development. Note that the Kanban board is similar to the "information radiators" of Agile methodologies [16], but it is not the same thing.

To be eligible to use the Kanban approach, the software development must satisfy the two Corey Ladas' postulates [17]:

1 *It is possible to divide the work into small value adding increments that can be independently scheduled.* These increments are called Features, User Stories, Stories, Work Items, or Minimum Marketable Features (MMF). The project ends when all its features are completed.

2 *It is possible to develop any value-adding increment in a continuous flow from requirement to deployment.* So, the work on each feature, from its specification to its deployment, is accomplished through a set of activities, performed in the same sequential order.

Initially, the features are put in a Backlog, from which they are pulled to be assigned to the first activity. When a feature is done in the last activity, it is automatically pulled from it, to a list of *"Live"* or *"Released"* features. The work is performed by a team of developers. Each developer has different skills related to the various activities.

When the work of a developer on a feature ends, or in any case at the end of the day, he looks for another feature to work at. Kanban systems focus on a continuous flow of work, and usually do not employ fixed iterations. Work is delivered as soon as it's ready, and the team only works on very few features at a time, to limit WIP and make constant the flow of released features throughout the development.

The growing interest on Kanban software development is demonstrated by the publication of various books, and by the proliferation of Web sites on the subject in the past couple of years. The most popular among these book was written by David J. Anderson [3]. Another book by Corey Ladas is about the fusion of Scrum and Kanban practices [17]. A third book on the subject was written by Kniberg and Skarin [18], and is also availabile online. The reader interested to a more detailed description of the Kanban approach should refer to these books.

6.1.2 Calibration of the model

The simulation model and its implementation, presented in this section, represent the first and basic version of the aforementioned model. In this case study we reproduce

a simple development process structured into four activities and with a team of then developers, skilled and involved in the different activities.

In the case of the Kanban process, the obvious parameters to optimize are the activity limits – the maximum number of features admissible in each activity. To this purpose, we devised a paradigmatic setting of a typical project, and used the simulator to assess the optimum limits. The settings common to all optimizations are the followings:

1 There are 100 features to be completed. The effort needed for each feature is a random number drawn from a Poisson distribution with average of 5 days. features with zero effort are discarded. The features are the same for all optimizations.

2 There are 4 activities, as described in Table 6.1. The actual effort needed to complete an activity on a feature is not perturbed, so $y_{i,k} = x_{i,k}$.

3 The feature limits on the four activities are chosen between 1 and a maximum value which is 9, 12, 7 and 4 for activities 1, 2, 3 and 4, respectively. Let us denote M_k the limit for $k-th$ activity, thus $M_1 \in 1,2,...,9$, $M_2 \in 1,2,...,12$, and so on. An exhaustive search can be done by performing 3024 simulations, one for each possible combination of the four parameters.

4 The number of developers is 10. Developers may be skilled in just one activity, or in all 4 activities, depending on specific runs performed.

5 The penalty factor p is varied between 1 ($nopenalty$) and 3 (in the case of change of feature the time needed to complete it is 3 times longer).

6 The cost function to minimize, $f()$, is the sum of the time, t_c, needed to complete all 100 features, and the weighted sum of the limits:

$$f(M_1, M_2, M_3, M_4) = t_c + w \sum M_k$$

where t_c is function of $M_1, .., M_4$, and the second term aims to minimize the limits themselves. The optimizations are performed for different values of w, typically 0 (no influence of limits), 0.2 (small influence), 1 or 2 (strong influence). Note that the order of magnitude of t_c is typically one hundred (days).

7 Time t_c is obtained averaging the project end time of 20 simulations, because tc varies randomly for each performed simulation. The average on 20 runs stabilizes it substantially.

6.2 Case Study Two

In this section we explain 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 data gathered from this case study to assess the goodness of 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.

Table 6.1: *The main features of the simulated activities*

Activity	Limit	% of total effort	Skilled developers
1.Design	3	30%	3
2.Development	4	40%	5
3.Testing	3	20%	3
4.Deployment	2	10%	2

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 Anderson in the chapter 4 of his book [3], 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% [55]. 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.

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.

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.

The Scrum process

Scrum is by far the most popular Agile development methodology [65]. 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(see fig. 6.1).

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

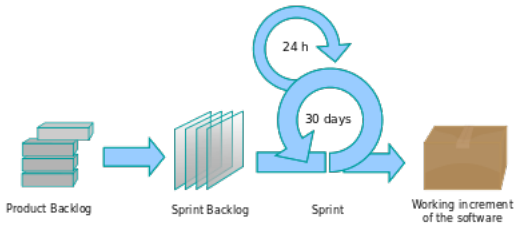

Figure 6.1: *The Scrum process*

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. The Kanban system was not seen in this way, because PSP was not replaced but merely augmented with the Kanban system.

6.2.1 Description of the approach

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 [62]. 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) [64]. 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 requests 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, according to known distributions of arrival times, priorities and efforts. 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.

6.2.2 Calibration of the model

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 in 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.

The model of the Kanban process

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. Note that 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 not heavily loaded and with a lot of slack capacity. Consequently, 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.

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.

6.3 Case Study Three

In this section we explain a real software development case study, where a transition was made from a traditional software engineering approach to a WIP-limited Lean approach. We use data gathered from a Chinese IT company to assess the goodness of the software process simulator we developed.

6.3.1 Case of 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[2]

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 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 6.2.

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

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

Table 6.2: *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: unspeciÂfied.
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)

their "Answer Date" and "Verify Date", while OPEN issues have often not set their "Answer Date" and "Verify Date".

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 was a significant number of issues reported during this period. In Figure 6.2 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 6.2 that the team devoted to resolve issues was not able to keep up with the pace of issue arrival for about two years (more than 700 days), until the number of issues waiting to be solved (Work in Progress, or WIP) reached the num-

ber 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 6 issues/day to about 3 issues/day.

To get 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.

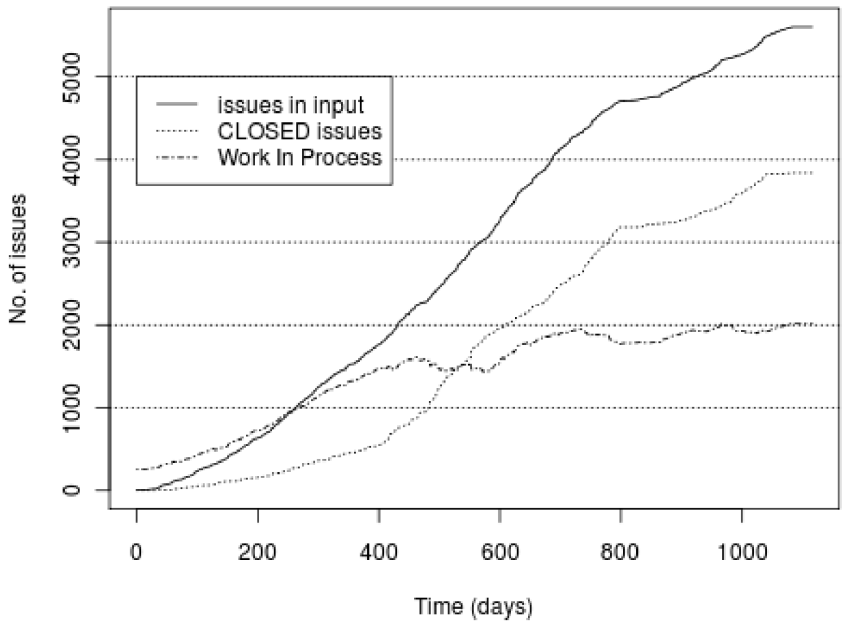

Figure 6.2: *Cumulative no. of issues in the system.*

We found that the time needed to manage the issues exhibits large variations, and is often longer than one 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 6.3 shows the main statistics of the considered times for closing, answering and verifying issues.

6.3.2 Description of the approach

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

Table 6.3: *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

- 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.
- Development (Analysis-Coding): 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.
- 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 6.3 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.

The "work items" in a maintenance project are the bug-fixing or enhancement requests, that we call "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 section). 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 one-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.

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), which accounts for variability in priority management. The lowest priority issue is assigned with a probability of being picked up proportional to 1, while the highest priority one is assigned with a probability proportional to $pm \geq 1$. 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 when there are no issues in that activity, or at the end of the 8-hour 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 due to 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 probability equal to 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, which 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.

Table 6.4: *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

6.3.3 Calibration of the model

We analyzed the total time required to manage issues. We are 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 get is as follows:

- in the beginning of the examined period, both teams were quite small (2-3 developers, plus a few part-time testers);
- as the number of unsolved issues became critical, about at day 400, more developers were added (up to a number of 7-8 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, is that the verification time is very small with respect to the analysis and coding time, so we concentrated to estimate the size and working time of the maintenance team. We can observe that when the maintenance team cannot keep up the pace of input issues, the size of work

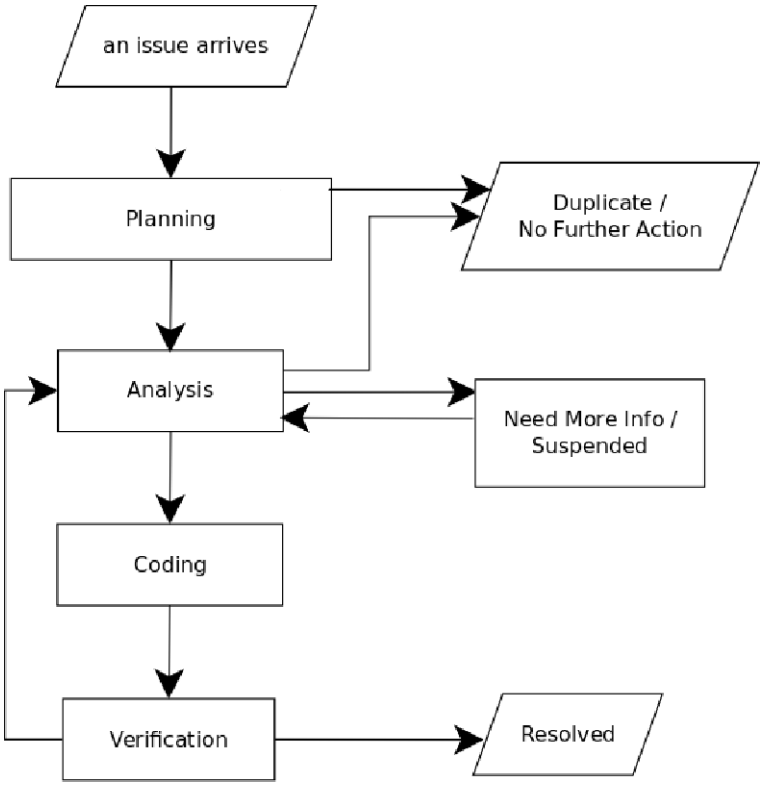

Figure 6.3: *A Generic Simulation Model for Software Maintenance Process.*

in progress increases, and the number of issues resolved depends only on the team's capacity, and not on the issues in input. From Figure 6.2, it is apparent that the number of issues completed follows three different patterns:

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 due to 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 the team size. 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 6.2. 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) with respect to the closed issues (3839). Table 6.4 summarizes the hypotheses on team size and on the average actual work needed to complete issues in the examined periods.

6.3.4 Case of Microsoft Maintenance Project

This section is dedicate to show the same case study presented in sec 6.2 but making a comparison between the original and the WIP-Limited processes. Some tables and figures are re-called from the previous sections.

Description of the approach

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 is worked on as follows:

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 hours 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 [60], 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 5 to 7 per month, with an average of 6. This meant that the "backlog" was growing up about 6 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.

WIP-limited 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. As described in [36], it was one of the first applications of the WIP-limited software maintenance process. To fix the performance problem of the team, a Lean-Kanban approach. 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 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.

A new process was devised to eliminate the waste. To limit the work-in-progress (WIP) and pull work from an input queue as current work was completed, were the main changes. They consideres 8 requests as WIP limits in development as well as 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.

6.3.5 Calibration of the model

Original Process

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. To model the original process, we introduced at the beginning of each day a check of the new

requests, creating a new event "StartDay". 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.

An important concept related to the work on requests is the *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 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:

$$t = \frac{t^{'} s}{p} \tag{6.1}$$

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:

$$q = \begin{cases} 1 & \text{if } n \leq 20, \\ \frac{20}{n} & \text{if } n > 20. \end{cases} \tag{6.2}$$

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.

WIP-limited Process

The presented 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, while 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 9. 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).

6.4 Case Study Four

Software project Risk management is crucial for the software development projects. Agile methodologies reduce risk using short iterations – and the consequent frequent user's feedback – feature-driven development, continuous integration. However, the risk of project failure or tine and budget overruns cannot be ruled out also in agile development. Process simulation, when applicable, is an important Risk assessment methods. In this paper we present a new approach to modeling some key risk factors and simulating their effects on project duration and time to implement features, using an enhanced version of our software process simulator. We studied and analyzed the critical aspects of software development risk that are suitable to be simulated. We then comparatively studied Scrum and Lean-Kanban processes in order to evaluate the presented Risk assessment method. This also resulted in a comparison of the two processes under the Risk management perspective, showing that Lean-Kanban looks more suited to manage errors in feature estimation, and the need of reworking a percentage of features that do not pass the software quality tests.

6.4.1 Description of the approach

While the Risk assessment methodology we are working on will include also other tools, such as interviews and risk-mitigation meetings, in this paper we will focus on the use of the developed simulator, that consistutes the most innovative aspect of the methodology. Our starting point in Risk assessment are the Six dimensions of risk, as defined by Wallace et al. [32].

They are:

1. Organizational Environment Risk, including change in organizational management during the project, corporate politics with negative effect on project, unstable organizational environment, organization undergoing restructuring during the project ,
1. User Risk, including users resistant to change, conflict between users, users with negative attitudes toward the project, users not committed to the project, lack of cooperation from users .
1. Requirements Risk, that is continually changing system requirements, system requirements not adequately identified or incorrect, system requirements not properly defined or understood.
1. Project Complexity Risk, encompassing high level of technical complexity, the use of new or immature technology.
1. Planning & Control Risk, including setting of unrealistic schedules and budget, lack of an effective project management methodology, project progress not monitored closely enough , inadequate estimation of required resources, project milestones not clearly defined , inexperienced project manager .
1. Team Risk, including inadequately trained and/or inexperienced team members, team member turnover, ineffective team communication.

Among these dimensions, the Organizational Environment Risk is out of the scope of this work. Many aspects of User Risk, Requirement Risk and Team Risk are specifically addressed by Agile Methodologies, that were introduced precisely for this scope. Our quantitative approach using SPMS addresses mainly dimensions 3, 4 and 5, and specifically inadequate estimations of requirements, project complexity in terms of number and estimated effort of requirements, and poor quality of software artifacts, due again to requirements not properly understood, or to issues in project management and planning. The Risk-assessment methodology we propose is performed in subsequent steps:

1. The development (or maintenance) process is modeled (activities, team, process, features, constraints) and the simulator is configured to simulate it.
2. Key quantitative risk factors are identified; in our case they are variations in estimated efforts to complete features or resolve issues, percentage of rework forecasted, variations in the skills of team members, probability of events that stall the development of single features, or block one or more developers, and so on.
3. Probability distributions are given to these key risk factors, for instance the probability distribution of actual effort needed to fix an issue, or the probability that a developer is blocked in a unit of time, together with the probability distribution of the time length of this block.
4. Key process outputs are identified, such as project total time, throughput, average and 95% percentile of lead and cycle times to resolve and issue, cost (obtained knowing the actual daily cost of developers).
5. Hundreds, or thousands of project Monte Carlo simulations are made varying the risk factors accordingly to their probability distributions, recording the key outputs.
6. The resulting distributions are analyzed, assessing for instance the most likely duration and cost of the project, the average time – or the distribution of times – to implement a feature or to fix a bug, the probability that a given percentage of features is implemented within a given time. Such Monte Carlo assessment might be performed also on an ongoing project, simulating the continuous flow of new requirements or maintenance requests, or just the remaining features to implement.

6.4.2 Calibration of the model

We tested the Risk assessment methodology on a fictitious case, representing a standard project, not too small and not too big, in order to be able to perform many Monte Carlo runs on it. To perform this kind of simulation run a simulation model was used. This simulator is event-driven and able to represent various kind of processes, the related activities, the features of the project and the developers with theirs skills and experience the important events that occur at any given time; it was presented in detail by the authors in their previous works [50] [51]. In this case it is customized to be able to represents the typical aspects of risk management using the Scrum and Lean-Kanban approaches,

Its requirements are 100 given features, representing functional requirements of the system to implement. Each feature has a given total effort estimation, in man-days, extracted from a Gaussian distribution with average $a = 2$ and standard deviation $s = 0.5$. The total average effort is thus $t = 200$ man-days, or about 9 man-months. Each feature has also a priority randomly chosen between 1 and 10. The activities involved in the project are Analysis, Implementation (coding), Testing and Deployment, accounting for 15%. 50%, 25% and 10% of the total effort, respectively. These activities can have an upper limit on the number of feature under work (WIP limits).

The development team is composed of seven people, with mixed skills. Each activity can be performed by more than one developer, but each developer is fully productive only in one activity. The other activity they are able to perform, if present, has a productivity reduced to 70 percent. This means that the time to complete a task in this activity is equal to the time needed by a 100% productive developer, divided by 0.7, that is multiplied by about 1.43. The team characteristics are shown in Table 6.5.

Table 6.5: *The composition of the case study team, with primary and secondary activities of developers. A: Analysis, I: Implementation; T: Testing; D: Deployment.*

Activities	Developer						
	1	2	3	4	5	6	7
Primary activity	A	I	I	I	T	T	D
Secondary activity	I	A	A	NONE	NONE	D	T

In the ideal case of all developers working 100% of their time in their primary activity, the project, whose total average effort is 200 man-days, would be completed in 200/7 = 28.6 working days. In practice, one has to account the initial and end slacks (when testers and deployers, and analysts and implementers have no feature to work on, respectively), the fact that often the work is performed in the secondary activity, with reduced productivity, and the time needed to study the feature when a developer works on it the first time, and when s/he switches from another feature, modeled using the penalty factor $p = 1.3$ in the simulator(see [50], section 3.2).

A feature, once Testing has been performed, can be judged of poor quality. In this case, it is sent back to Analysis to be reworked. When a feature is sent back to Analysis, WIP limits of Analysis can be temporarily overcome. In the case of rework, the effort, q, to perform again on the feature in Analysis, Implementation and Testing is 50% of the original effort, because a substantial part of the work was already performed. This holds also if a feature does not pass Testing more than once.

If r is the rework probability, and $q = 0.5$ is the rework, it can be demonstrated that the overall average work, w, on each feature is given by:

$$w = 1 + q\left(\sum_{i=1}^{\infty} r^i\right) = 1 + q[\frac{1}{1-r} - 1] = \frac{1-r+qr}{1-r} \tag{6.3}$$

We tested the methodology on two different agile processes â Lean-Kanban and Scrum. Lean-Kanban is characterized of a continuous flow of work on features, in order of priority but with a limits on WIP. We set the limits according to the number of available

developers for each activity, plus a small slack. The maximum number of features under work in each activity are: 3, 6, 4 and 2 for Analysis, Implementation, Testing and Deployment, respectively. There are no meetings scheduled at specific times, and the short Stand-up Meeting hold every day is not explicitly accounted for. The completed features are immediately released. The work on the project ends when the last feature is released.

The Scrum process we tested is characterized by iterations of two weeks (10 working days), with a post-mortem meeting after each iteration, and a iteration planning meeting before each iterations. The cumulative length of these two consecutive meetings is considered of one day the short Scrum meeting hold every day is not explicitly accounted for. At the beginning of each iteration, features non overcoming 50 man-days of cumulative effort are chosen in order of priority to be implemented in the iterations. If one or more features are not finished after the iteration, they are send to be completed in the next iteration. The completed features are released at the end of the iteration. When all features are completed, the last iteration is stopped and the work on the project ends.

The applying of the proposed Risk assessment methodology to the case study is outlined in the following steps.

1. The two development processes are modeled as described above. All other parameters used in the model are taken from the simulation of real cases, coming from industrial cooperations of our research group.
2. The key risk factors identified are:
   * the variations in estimated efforts to complete features;
   * the percentage of rework needed to make the feature pass the Testing phase.

   Variations in the skills of team members, probability of events that stall the development of single features, or block one or more developers are not considered for the sake of simplicity, though the simulator could account also for these factors.

3. For each identified risk factor, the probability distributions are
   - for each feature, the effort estimation variations follow a log-normal distribution with average the original effort and given standard deviation; we chose the log-normal because it guarantees that estimation remain positive, and is balanced in terms of percentage variation; the value of the standard deviation is varied from zero to 5 man-days (we used the values 0, 1, 2, 3 and 5), to assess various levels of risk;
   - the probabilities of rework after Testing are the same for each feature; they are varied from zero to 50% in steps of 10%, as above.

4. The key process outputs whose variations are checked are:
   - project total time, from project start to the time when the last feature is released;
   - this is inversely proportional to throughput;
   - statistics on lead and cycle times to implement a feature; they are:

    * average length, measuring the average time to give value to the customer;
    * standard deviation, measuring variations in the times;
    * median time, measuring the most likely time to complete a feature;
    * 95% percentile of times, measuring the limit of the worst 5% times;
    * maximum time. cost (obtained knowing the actual daily cost of developers). For each relevant output, Risk.

   On each of these values it is possible to set Risk thresholds that, if reached or overcome, trigger proper mitigation actions.

5. We performed 100 Monte Carlo simulations for each choice of the tested risk factors, recording the key outputs.

Chapter 7

Experimental Results

7.1 Simulation Results

In this section we presented the experimental results obtained from the analysis of different case studies. The main goal of this research work was to investigate the effects of the use of a WIP-Limited approach applied to a real project. Before, we studied the project, analyzed the data collected and the process followed. Then, using data collected from the various projects, as input to the simulation model, we performed many simulation runs in order to better understand the processes and their dynamics. In fact the simulation approach is useful to analyze the effects of the use of a WIP-Limited approach in comparison with other kind of development or a maintenance processes.

7.2 Results Case Study One

We performed some preliminary simulation to assess the overall model, and to highlight the differences between a limited-WIP and an unlimited process. The settings were those of a typical average software project, with an initial backlog of 100 features, and 40 features added during the development. The features' efforts are drawn from a Poisson distribution, with average equal to 5 man-days and standard deviation of 2.2 man-days. So, the total amount of man-days of the project is 700 (140 x 5). There are 4 activities, to be performed sequentially, shown in Table 6.1 in chapter 6.

Note that the description of the activities is conventional. Any other description will yield the same results, provided that the limits (max. nr. of features) and percentage of total effort is the same. When a feature is pulled to an activity, the actual effort to complete it, $y_{i,k} = x_{i,k}$ r, as described in Section 6.1, is computed. The Lognormal distribution used has mean equal to 1 and standard deviation equal to 0.2-meaning that deviations of more than 20% from the theoretical effort are common. The team is composed of seven developers. Most of them are skilled in two activities, so the total number of skilled developers in 6.1 is greater than 7. For the sake of simplicity, all skills are set to one, and there is no skill improvement as the development proceeds. The penalty factor p of eq. 5.2 is set to 1.5. We performed the following simulation tests:

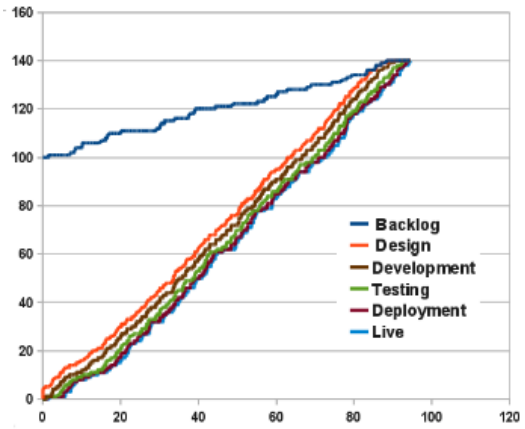

Figure 7.1: *WIP Limits, devs with one or two skills*

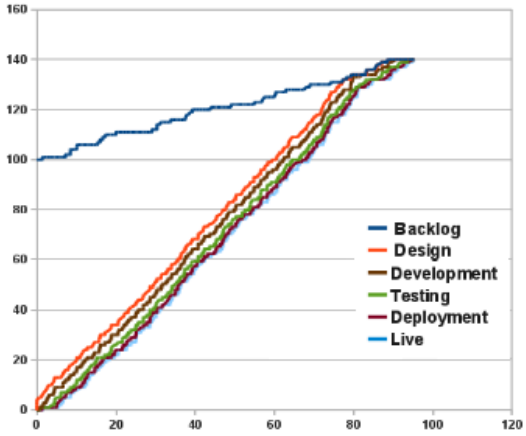

Figure 7.2: *WIP limits, devs skilled in all activities*

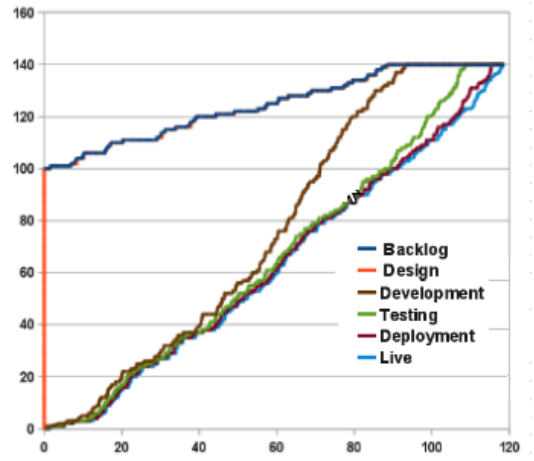

Figure 7.3: *No limits, devs with one or two skills*

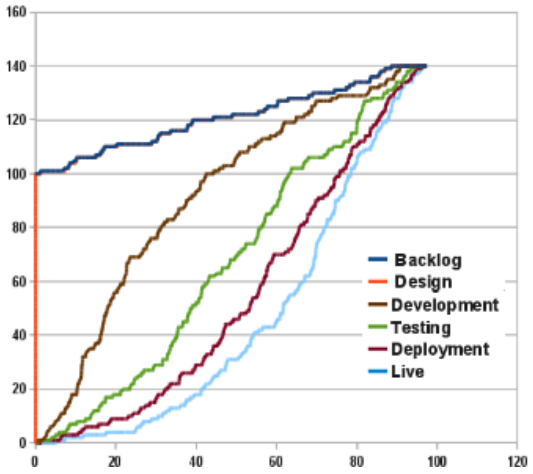

Figure 7.4: *No limits, devs skilled in all activities*

1 setting as described above;

2 settings as above, but with developers skilled in all activities;

3 settings as above, but with no feature limits in the activities (unlimited WIP).

4 unlimited WIP, with developers skilled in all activities.

The resulting Cumulative Flow Diagrams (CFD) of the four simulations are shown in Fig. 7.1 7.2 7.3 7.4. The times (horizontal axis) are in days of development. As you can see, the diagrams in the cases of limited and unlimited WIP look very different â in the former case, there is an almost constant flow of features that are completed, while in the latter case the CFD is much more irregular. Another difference is between the case when developers are skilled just in one or two activities (see Table 6.1), and the case when developer can work in all four activities with the same proficiency (this situation is unlikely in the real world). In the former case, developers can work only to one or two activities, meaning that

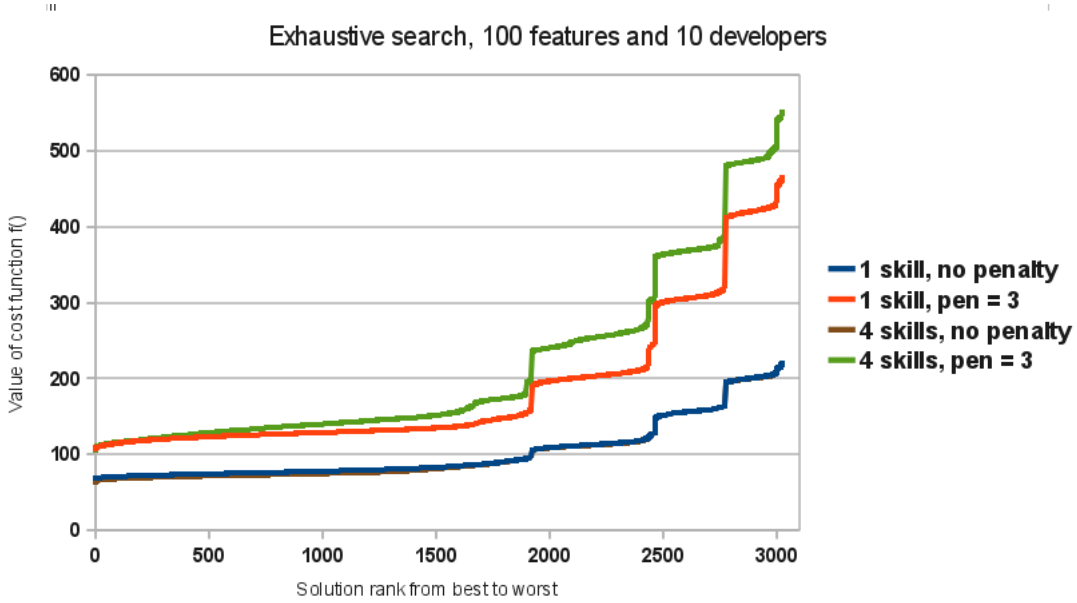

Figure 7.5: *Plot of the cost functions in ascending order for 4 exhaustive searches, with different values of the penalty and of the developers' skills.*

Table 7.1: *The four best points found in the search, for the four cases considered*

	no penalty		**penalty=3**	
Best points	**1** skill	**2** skill	**3** skill	**4** skill
1	**3,6,4,1**	**4,5,3,2**	**3,4,3,1**	**3,5,3,2**
2	**3,5,4,1**	**4,6,3,2**	**3,5,3,1**	**3,6,3,2**
3	**3,6,3,2**	**5,5,3,2**	**3,5,2,2**	**3,7,3,2**
4	**4,5,4,1**	**5,6,3,2**	**3,4,4,1**	**4,4,3,2**

for instance at the beginning of the project the testing and deployment developers are idle, while designers are idle at the end of the project. This situation enforces some limits on the work that can be done, even when activities are not limited, and is reflected in Fig 7.3, which show a CFD more structured than in the case of general-purpose developers. On the other hand, in this case the efficiency of the whole process is lower, and the time to complete all features is more than 20 longer than in the other cases. When developers work on every activity, and are thus never idle, the length of the project tends to be shorter.

In general these simulations confirm the validity of the WIP-limited approach, and give insights on what really happens under different settings of the development.

7.2.1 Optimization of the activity limits

We, also, performed various optimizations, performing exhaustive searches after changing some parameters. Note that, in the case of more activities, wider search intervals, or more complex simulations, it would be easy to perform the optimization using minimization al-

gorithms in place of the exhaustive search. To test the stability of the optimization, we computed the average, the standard deviation and other statistics of 20 different computation of the cost function (each obtained by averaging 20 simulation runs), for ten different values-randomly chosen-of the vector $M = (M1, M2, M3, M4)$, with a simulation referring to a penalty factor of 1.5, and to 10 developers with different skills. The weight w was set to 0.2. The standard deviation was always between 0.2% and 0.3% of the average, thus showing a substantial stability of the results. To demonstrate the feasibility and the power of the approach, we show an example of result in Fig 7.5. It represents 3024 runs (exhaustive search), with $penalty = 1$ (no penalty) and 3 (high penalty to change feature). Cost function has $w = 1$. Developers can be with no specific skill (they can work on all 4 activities), or with just one skill. In the latter case 3 developers are able to work on activity 1, and 4, 2, and 1 developers are able to work on activities 2, 3 and 4, respectively. (The number of developers able to work in each activity is proportional to the activity's relative importance in the overall effort.)

The results are in ascending order of the cost function $f()$. The *"jumps"* represent limits able to delay the whole development, for instance a limit of 1 or 2 features in activity 2. The case with no skills is heavily penalized when $penalty = 3$, because developers can work in every activity, and are more prone to change the feature they work on in the next day, thus taking the penalty. If developers are constrained to work only on a single activity, this change is less likely (and even less likely when WIP is low). Table 7.1 shows the best 4 points (values of limits M_1, M_2, M_3, M_4) for the 4 cases optimized.

7.3 Results Case Study Two

We simulated the three processes presented in 6.2 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.

7.3.1 The original process

Fig 7.6 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.

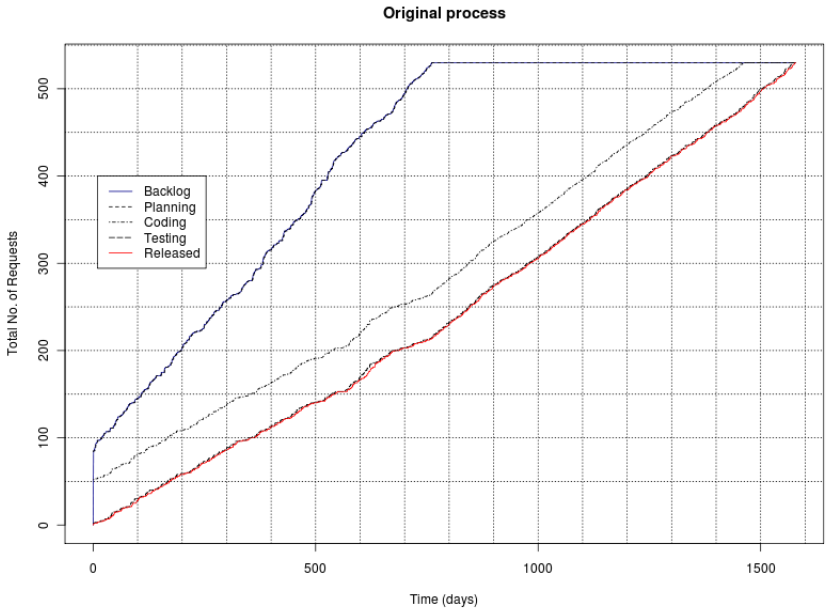

Figure 7.6: *The CFD of the original process.*

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 7.2 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

Table 7.2: *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

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.

7.3.2 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.

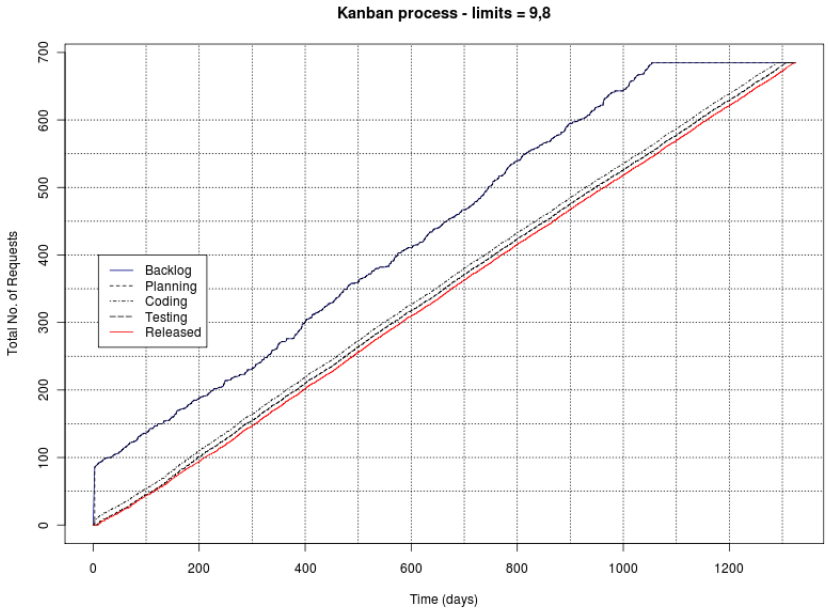

Figure 7.7: *The CFD of the Kanban process.*

The resulting CFD is reported in Fig. 7.7. 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 Kanban system, the throughput substantially

Table 7.3: *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

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 (see for instance [54]). 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.

Regarding cycle times, their statistics are shown in Table 7.3, 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.

7.3.3 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 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

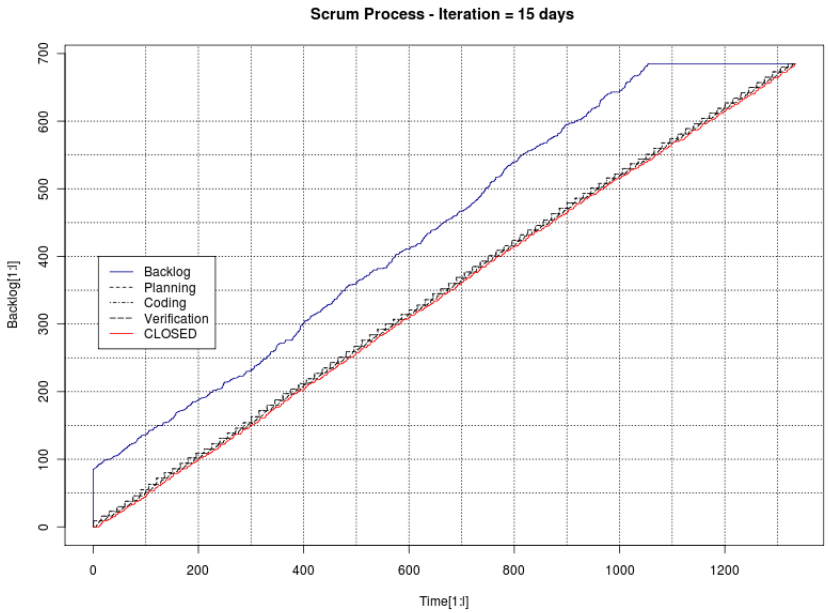

Figure 7.8: *The CFD of the Scrum process.*

Table 7.4: *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

fact, with 14 working days per team member during a Sprint, we have a total of 84 man days. We slightly increased this limit to 90, to accommodate 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. 7.8 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 7.4. 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.

7.4 Results Case Study Three

Using the proposed simulation method described in Chapter 6 Section 6.3 and the data described in Section 6.3 we performed simulations for the original (without WIP-limits) and Lean-Kanban (with WIP-limit) processes.

7.4.1 Results of the Chinese Firm case

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.

Simulation of the existing process

We first adapted the generic simulation model for software maintenance process as described in Chapter 6, so as to simulate the process reflected by the data presented in Section 6.3.1. The adapted process has the following characteristics:

- 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.
- The effort of each issue is drawn from a distribution mimicking the distributions shown in Table 6.4. 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 hours 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 6.4.
- The maintenance phases have the following specific characteristics:

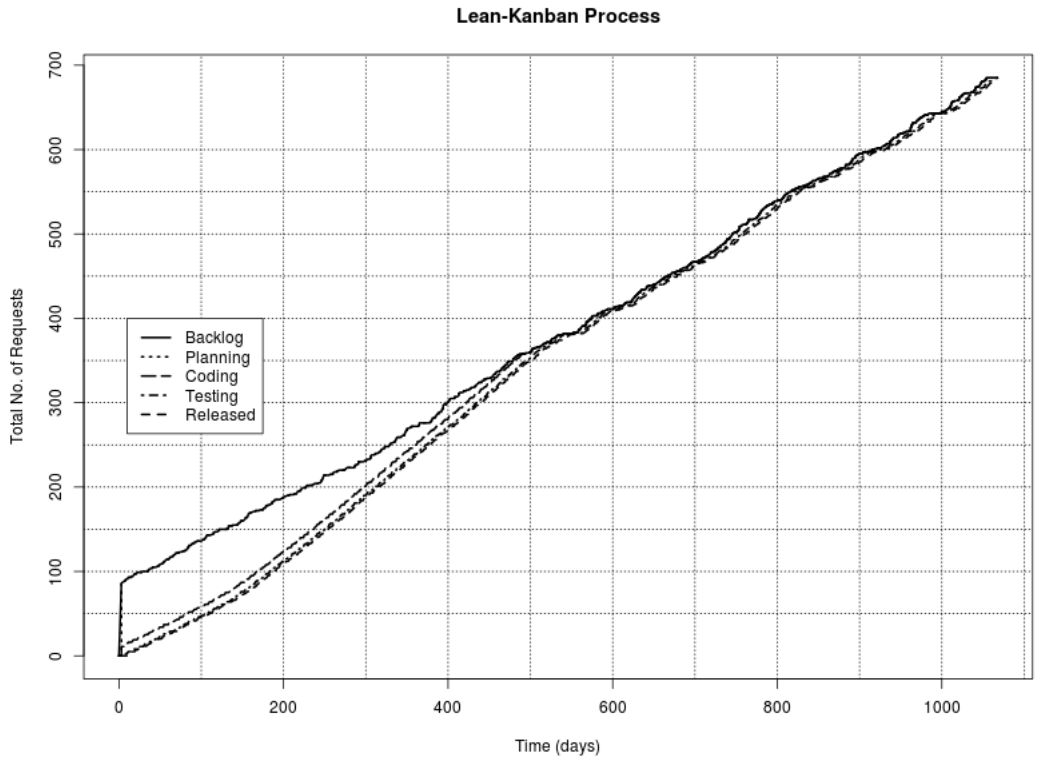

Figure 7.9: *The WIP diagram of the WIP-limited process with a developer and a tester added after six months.*

  - **Planning**: effort in this phase is considered negligible, since 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 zero or one 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.

- 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 [54]. However, changing these percentages does

not have a substantial impact on the results, provided that the Verification quota does not increase.

- 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 6.3, and specifically in Table 6.4, we considered three time intervals, whose length is 400, 400 and 284 days, respectively. They cover all the considered period of 1084 days. Table 6.4 reports, among other information, the number of developers devoted to maintenance and verification during the various phases.

- The factor pm (introduced in Section 6.3, 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.

- 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, ta 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 7.10 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.

Table 7.5: *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

Overall, in Figure 7.10, what really matters is the dotted curve representing CLOSED defects, which shows a good match to the dotted curve in Figure 6.2. The two solid curves in Figures 6.2 and 7.10 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.

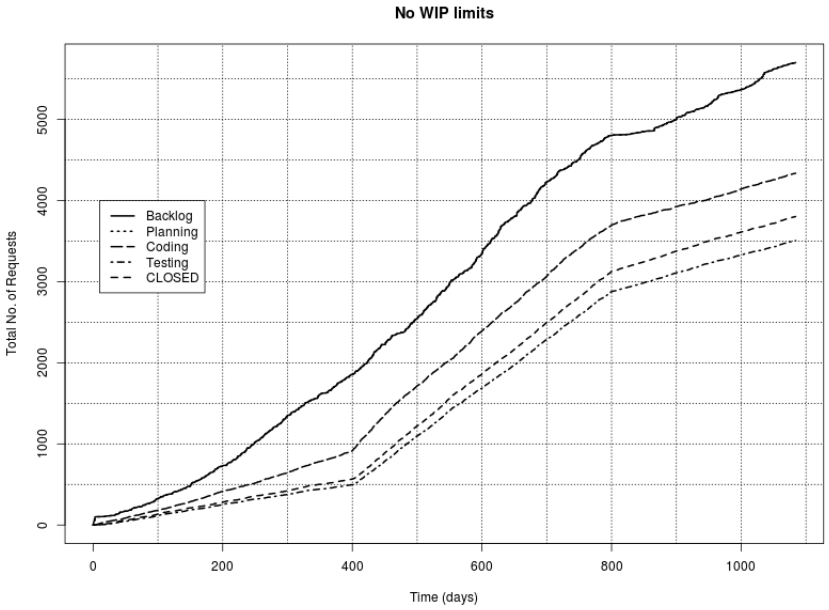

Figure 7.10: *The WIP diagram without WIP limits. The Planning curve overlaps with the Backlog curve.*

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) is 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 7.6.
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 roomavailable there (in order not to violate the 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.

Table 7.6: *Limits verified and actually used for the various activities during the simulation*

Statistics	Original data	Planning		Analysis		Coding		Verification	
		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

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

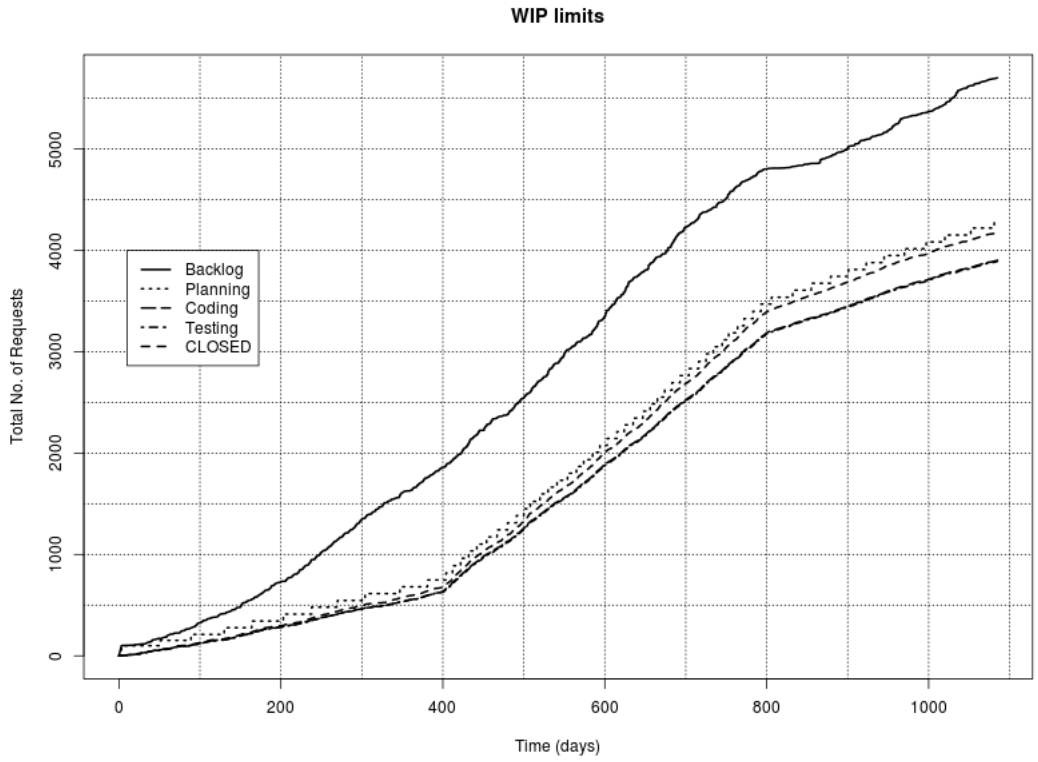

Figure 7.11: *The WIP diagram with WIP limits. The Testing curve overlaps with the Released curve.*

the number of issues belonging to it falls below 1/3 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 7.6.

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;

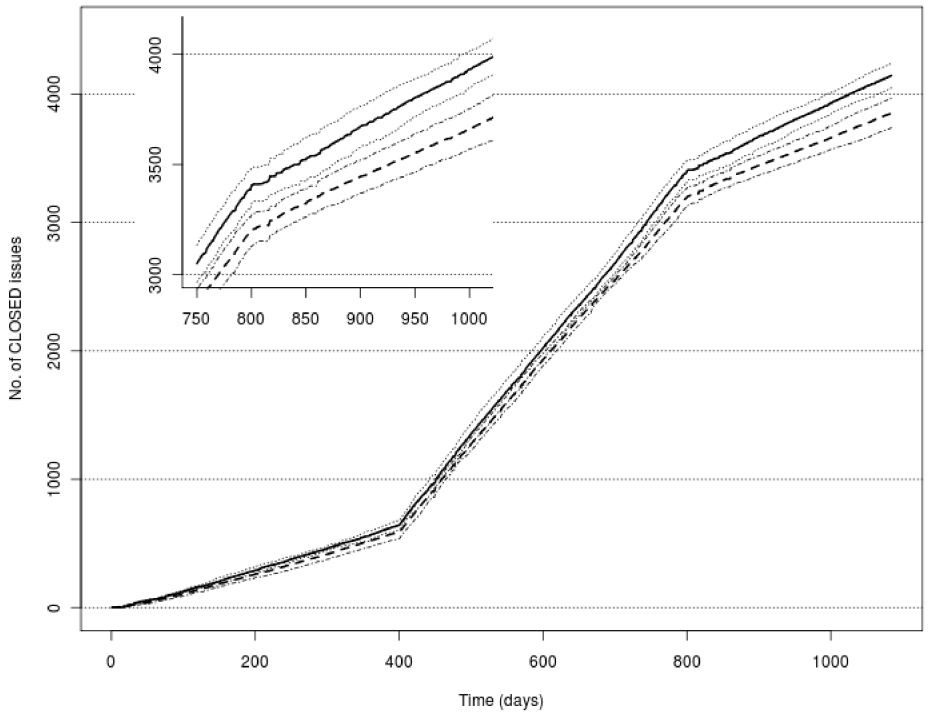

Figure 7.12: *Comparison of the number of CLOSED issues vs. time. The results are average of 10 runs, in WIP-limited case (solid line) and non-limited case (dashed line). The other lines represent two standard deviation intervals.*

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 7.6 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 7.11 shows the cumulative number of defects in the various possible states for the WIP-limited simulation. This figure differs from Figure 7.10, 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 number of system throughput.

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 7.12 we show the number of issues in the CLOSED state, the average of 10 runs, and their two standard deviation lim-

its. 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 due to the penalty that is applied when the work on an issue is resumed by a different developer.

7.4.2 Results of Microsoft case

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 four 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), while 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 6.1 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 subections we report the results of the two processes.

Simulation of the Original Process

Figure 7.13 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 6 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, while 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.

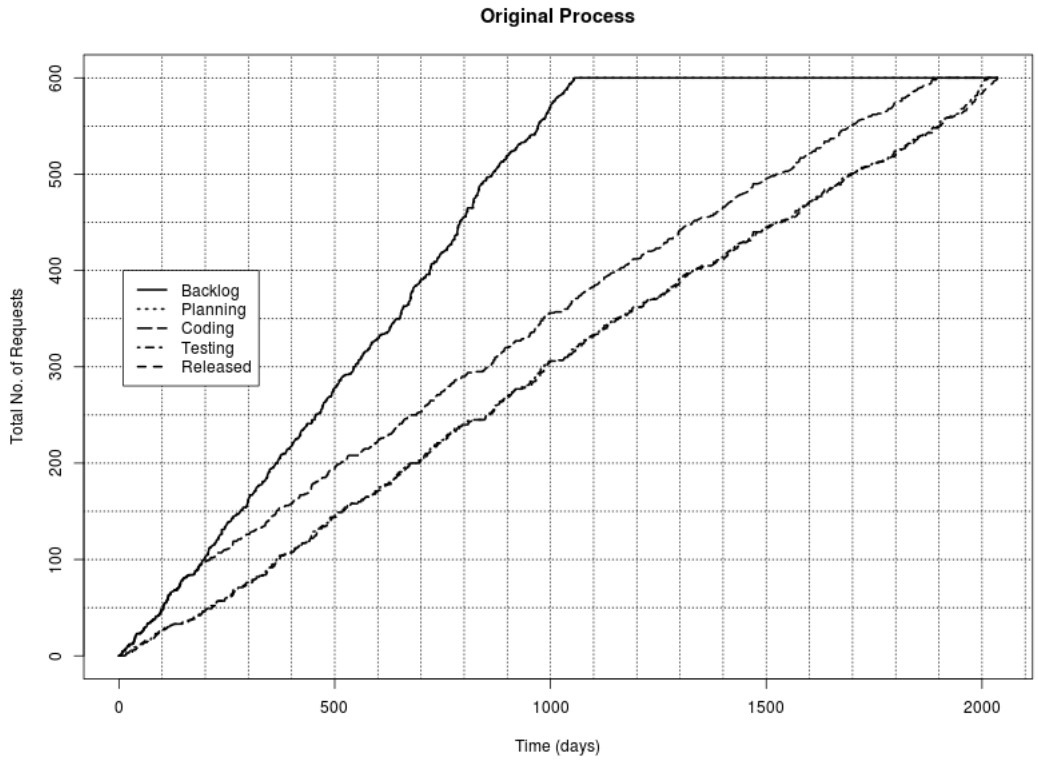

Figure 7.13: *The WIP diagram of the original process.*

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, 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 7.2 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.

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.

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 7.14. 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,

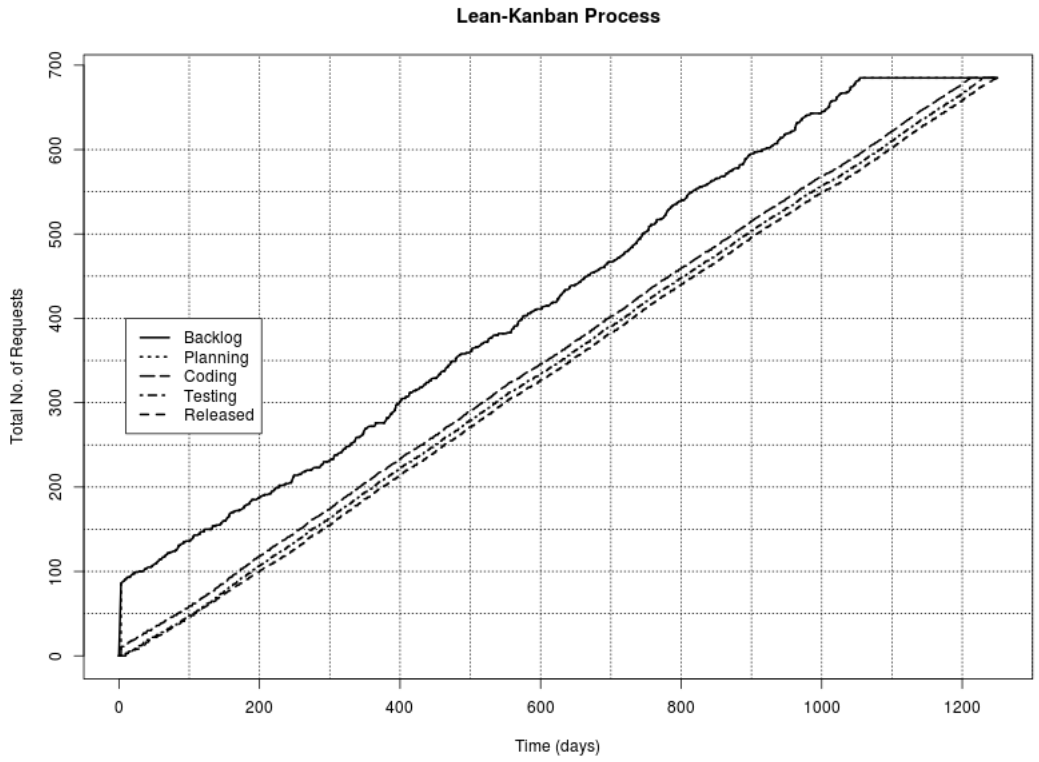

Figure 7.14: *The WIP diagram of the WIP-limited process.*

the throughput substantially increases compared withthe 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, due to 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, 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 [66]). 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 7.3. 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 7.14 shows the cumulative flow diagram in this case, showing that the backlog is reduced to zero in 300 working days, that 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.

7.5 Results Case Study Four

This section is dedicated to the discussion of the simulation results that come from hundreds simulations made in the previous section and considering different cases. Table 7.7 shows basic statistics of the 100 features used in the simulations, with no effort variation, and with the quoted Lognormal variation (in the latter case the values are averaged on all 100 simulations). The original data pertain to a Gaussian distribution with a finite number of items. The data with Lognormal variation show an increasing skewness of their distribution. The average is approximately unchanged, while the median shifts toward zero, and the tail values

Table 7.7: *Statistics of effort of the 100 features used. Data in case of Lognormal variation report the average over 100 different random extractions.*

Statistic	Original data	Standard Dev. of Lognormal Variation			
		1.0	2.0	3.0	5.0
Average	1.98	1.96	1.96	2.02	1.91
Std.Dev	0.45	1.08	1.90	2.78	3.67
Median	1.95	1.77	1.42	1.12	0.73
95% percentile	2.70	3.91	5.39	6.53	7.21
Maximum	3.20	5.99	11.57	18.77	26.37

shifts toward higher and higher values, as the tails becomes fatter and fatter.

We run 100 simulations for each choice of the Risk parameters, and for the two tested processes – Lean-Kanban and Scrum. Table 7.8 shows a comparison of the two methods in the no-Risk case, that is no rework and no variation on input feature estimation. The results are averaged on 100 simulations, with the same input data, but differing in the seed of the random number generator. The reported data do not vary much along the simulations, as highlighted by the low value of the standard error.

Lean-Kanban approach shows a superior performance in all parameters – shorter duration, lead and cycle times, and lower variations of these times. In particular, cycle times,

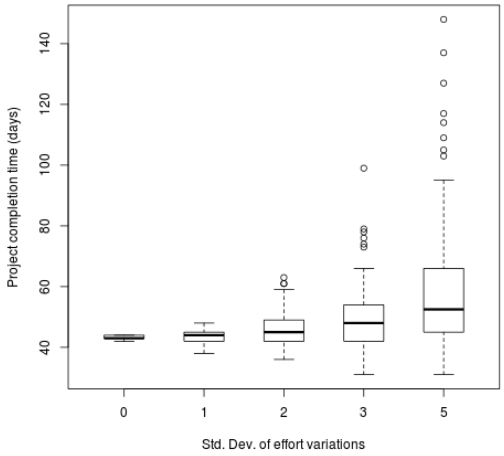

Figure 7.15: *Lean-Kanban Process. Box plot of project duration as a function of variations in features' effort. Each box refers to 100 simulations.*

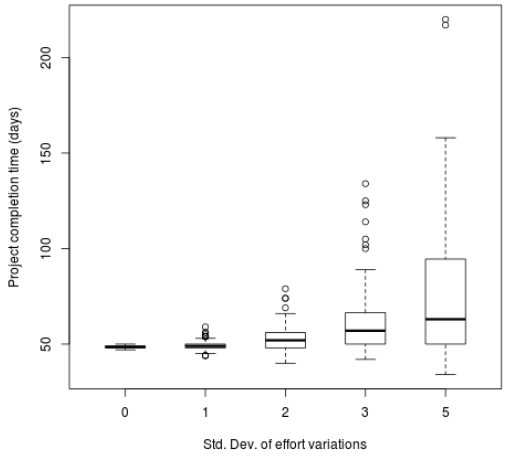

Figure 7.16: *Scrum Process. Box plot of project duration as a function of variations in features' effort. Each box refers to 100 simulations.*

representing how long it takes to implement a feature, from the start of the work on it to its release, are 2-3 times shorter, and much stabler.

Table 7.8: *Statistics on 100 simulations for each method, with no rework and no feature effort variation. The data shown are the average of the referred parameters. Within parenthesis we report the standard error*

	Lean-Kanban	**Scrum**
Project duration	43.26 (0.054)	48.56 (0.064)
Mean of Lead times to complete a feature	22.4 (0.022)	30.2 (0.028)
Std. dev. of Lead times	11.6 (0.011)	13.4 (0.024)
Median of Lead times	23.0 (0.035)	30.0 (0.0)
Mean of Cycle times to complete a feature	3.97 (0.006)	13.19 (0.022)
Std. dev. of Cycle times	0.92 (0.006)	4.68 (0.011)
Median of Cycle times	3.91 (0.008)	10 (0.0)

In Fig. 7.15 7.16 we report the box-plot statistics on the total project duration varying the standard deviation of features' effort, keeping their average at 2 man-days, and keeping the percentage of rework equal to zero. Each box shows the results of 100 simulations for the standard deviation value reported in abscissas. The image on the left refers to Lean-Kanban, showing a slowly increasing median value, together with an strong increase of total time variations. The image on the right refers to Scrum and shows a similar behavior, but centered around higher median values of project duration.

Fig. 7.17 7.18 show the box-plot statistics on the total project duration varying the percentage of rework, while keeping the features' effort standard deviation equal to zero. Each

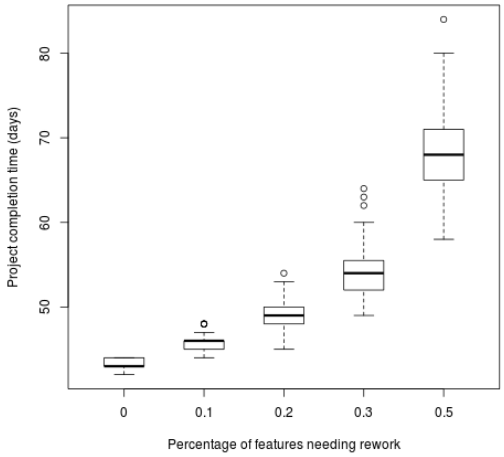

Figure 7.17: *Lean-Kanban Process. Box plot of project duration as a function of features percentage that don't pass the SQA phase and need rework, keeping.*

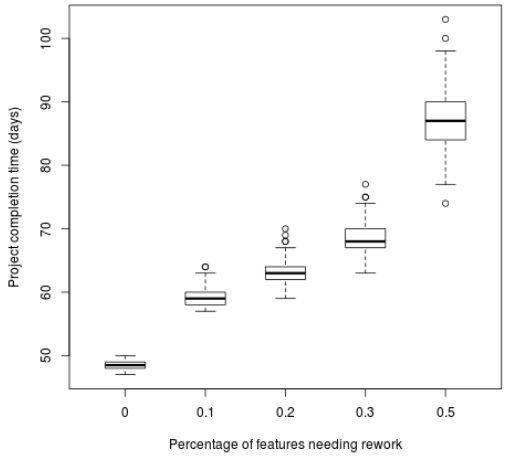

Figure 7.18: *Scrum Process. Box plot of project duration as a function of features percentage that don't pass the SQA phase and need rework, keeping.*

box shows the results of 100 simulations for the rework percentage reported in abscissas. The image on the left refers to Lean-Kanban, showing a strong increase both in the median and the variations as rework percentage increases. The increase is similar to that expected by multiplying the factor w of eq. 6.3, with $q = 0.5$ and r equal to the rework probability, by the base value of 43 (the median of the project duration in the case of no rework). The image on the right refers to Scrum. Here the increase in median effort and in volatility is even bigger. We note also a strong leap in the first step from no rework to 10% rework, that can be explained perhaps with the fact that even a small rework percentage is able to increase the number of features that are moved from an iteration to the next, thus altering the whole project schedule. In Fig. 7.19 7.20 we report the box-plot statistics on the total project duration varying both the percentage of rework and the features' effort standard deviation. This figure substantially confirms the results of Figs. 7.15 7.16 and 7.17 7.18. The last box, denoted by E, refers to the case with both the highest rework percentage and effort variation, so it exhibits the highest median and the highest variation for both Lean-Kanban and Scrum cases. As regards lead and cycle times, we will focus only on cycle time, because in the reported case all features enter the system at time zero, so lead times are not very significant. Fig. 7.21 reports the medians and the standard deviations of cycle times, varying the rework percentage and the features' variation, for both processes. Each point of the graph refers to the average made on 100 simulations. We use the median, that is more significant than the mean for this analysis, and for the sake of simplicity we show only a subset of the computed cases. In abscissas there is the percentage of rework, while different curves refer to specific processes and different standard deviations of the feature efforts.

Risk assessment

Starting from the Monte Carlo simulation results, it is possible to assess the risks of the project, with respect to variations in delivery time and cycle time. For the sake of simplicity,

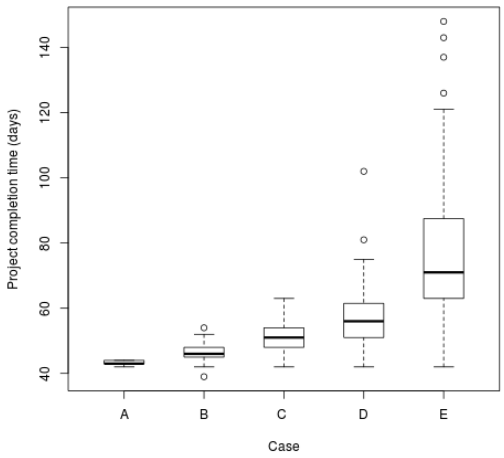

Figure 7.19: *Lean-Kanban Process. Box plot of project duration as a function of the percentage of rework and effort variation: A: no rework and no variation; B: 10% rework, effort Std. Dev. = 1.0;C: 20% rework, effort Std. Dev. = 2.0; D: 30% rework, effort Std. Dev. = 3.0; E: 50% rework, effort Std. Dev. = 5.0.*

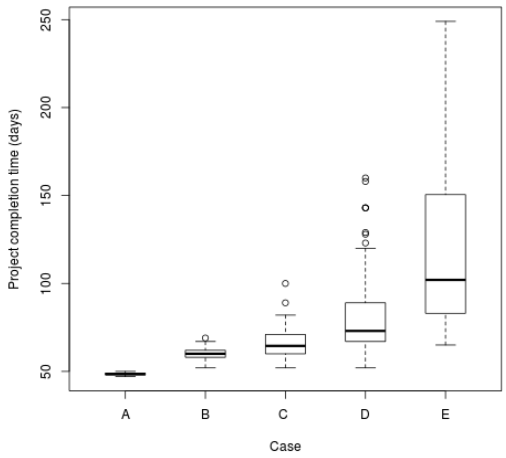

Figure 7.20: *Scrum Process. Box plot of project duration as a function of the percentage of rework and effort variation: A: no rework and no variation; B: 10% rework, effort Std. Dev. = 1.0;C: 20% rework, effort Std. Dev. = 2.0; D: 30% rework, effort Std. Dev. = 3.0; E: 50% rework, effort Std. Dev. = 5.0.*

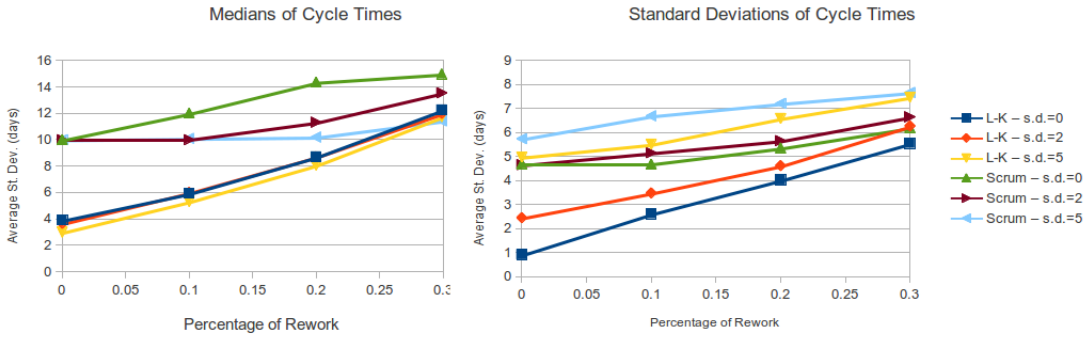

Figure 7.21: *Medians and Standard Deviations of Cycle Times, for various values oWhen both of the parameters. These quantities are averaged over 100 simulations.*

let us consider only Lean-Kanban process. In the case of no rework and no significant error on feature effort estimation, we can expect a very stable project duration of about 43 working days, and a cycle time distribution whose median is 4 days and with a standard deviation of 1 day (first dot of blue lines in Fig. 7.15 7.16). These figures are the starting point, that can be changed only by varying the process or the team composition and/or skills. In the case of an increase only in features' efforts, Fig. 7.21 shows that, while the average project length does non increase too much, but there are several instances of very long projects. For instance, in the case of Lognormal noise with standard deviation equal to 3, the 5% highest percentile of

project length is about 75 days, to be compared to an average length of 49 days. This means that there is 5 percent chance that the length of the project will overcome 75 days. In the case of standard deviation equal to 5, the 5% value at risk increases to 115 days, compared to an average length of 53 days. These figures are to be considered for risk mitigation during the project, if strong variations of feature efforts emerge. When rework is considered, clearly the project length increases with rework percentage, as shown in eq. 6.3. Fig. 7.17 7.18 show that the tails of length distribution are less extreme than in the case of effort variation. However, when rework percentage is substantial, the inherent effects on project length must be considered for risk mitigation. When both feature effort variations and rework are present, as shown in Fig. 7.19 7.20, the increase in average project length is roughly additive, while the length variations do not add up. As regards cycle times, the simulations show that feature effort variations have almost no impact on their median, while they obviously impact on cycle time variance. The rework, on the other hand, induces a strong increase in cycle time. A 30 percent chance of rework causes a 3-4 fold increase of the cycle time median. These results should be properly taken in account performing risk assessment, depending on actual values of effort variation, rework, and criticality of cycle time variation.

Scrum vs. Kanban

A by-product of our study is that it offered a comparative assessment of Lean-Kanban and Scrum methodologies, though on a simulated paradigmatic case. In fact, the issue of Kanban-Scrum comparison, and of migration from Scrum to Kanban seems to have raised strong interest in recent times [78] [77] [83].

Our results show that Lean-Kanban is consistently more efficient than Scrum as regards project length. This was expected, because on one hand Lean-Kanban has no *waste* time due to sprint planning meetings at each iteration, and on the other hand its continuous flow fully employs all developers in all activities, but for short transients at the beginning and at the end of the project. Conversely, each Scrum iteration entails such transients. One might argue that such waste of time and effort in Scrum is not realistic, because also Kanban needs meetings, and because during sprints team members specialized in activities such as analysis and testing will anyhow find something to do.

However, Sjoberg et al. claimed a 21% productivity gain migrating from Scrum to Kanban [78]. This should be compared with our 12% decrease in average project duration with no perturbations, that increases exactly to about 20% when more realistic effort uncertainty and rework factors are introduced.

So, we may conclude that our results are consistent with their results as regards average productivity. The situation is even more favourable for Kanban when risk factors are considered, because our simulations show that the amplification of average project length and variations due to the considered risk factors tend to be higher in Scrum than in Kanban.

Regarding cycle times, Sjoberg et al. report a 50% decrease of average lead times migrating from Scrum to Kanban [78]. Our simulations report an even higher 70decrease of average cycle times (see Table 7.8). In our case study lead times are not relevant, because all 100 features are given at the beginning of the project, and no new feature is added afterwards. Adding a constant waiting time to our cycle times would lower our 7% gain, yielding a value closer to that reported by Sjoberg et al. median and variance look more sensitive. This result on cycle times is obvious, because in Kanban features are released when actually finished, and not at the end of each iteration as in Scrum. When feature efforts variations and

rework are introduced, cycle times increase in both methods, but Scrum looks more resilient because the slack due to fixed lehght iterations accommodates at least partially the increase of cycle times.

To summarize, the comparison of Scrum vs. Lean-Kanban on the simplified case study used in this study confirms some empirical findings of another comparison made on a real case study. It shows that the simpler and less structured approach of Kanban seems to yield a better productivity, and a higher resilience with respect to risks as previously defined.

Chapter 8

Discussion of Experimental Results

In this section, we briefly discuss about the results obtained applying the proposed model to different real cases, reported in Chapter 7. The model applied to the different real cases is the same, but, it was customized in order to adapt it to represent different case studies, as we underlined in the section of model calibration. In previous Sections (Case Study 1 7.2, Case Study 2 7.3 Case Study 3 7.4 and Case Study 4 7.5), we showed, that this model was able to replicate quite well the empirical characteristics of the real maintenance and development projects studied.

From results of second case study, we obtained some statistics about cycle time in various time intervals of the simulation. In general, these statistics show very long and very variable cycle times. Around this time, the average and median cycle times are 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. In the case of Lean-Kanban Process, from the CFD it is possible to note the slight increase in the steepness of the Coding and Testing lines after a given day, with a consequent increase 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. In case of Scrum Process the cycle time statistics 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. 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. Overall, all the choices done during the model calibration phase seem broadly correct if we consider the results that are very close to the real ones. Indeed, as all the simulation models, also our model must be calibrated every time that we want to analyse a new process.

Regarding the results from the third case study, 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 and considering the WIP-Limited process, from the obtained result, it is possible to denote a very short lead time to work on defects. Overall, the total number of defects closed at any given day is 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 number of system throughput. We show the number of issues in the CLOSED state, the average of the number of runs,

and their two standard deviation limits . 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 due to the penalty that is applied when the work on an issue is resumed by a different developer.

If we analyzed, also, the WIP diagram for the data of the original process that come from the Microsoft case, the figures show the inability of the process to keep pace with incoming requests. We reported some statistics about cycle time in various time intervals of the simulation. In general, these statistics show very long and variable cycle times. Instead, in the case of WIP-Limited process we noted the slight increase in the steepness of the Coding and Testing lines after a specific instant, with a consequent increase of the contribution made by Testing to the WIP With the adoption of the Lean-Kanban approach and the throughput, substantially, increased compared with the original process. 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.

Regarding the analysis of case study four, the Lean-Kanban approach shows a superior performance in all parameters – shorter duration, lead and cycle times, and lower variations of these times. In particular, cycle times, representing how long it takes to implement a feature, from the start of the work on it to its release, are 2-3 times shorter, and much stabler. We analyzed statistics on the total project duration varying the standard deviation of featuresâ effort, keeping their average at 2 man-days, and keeping the percentage of rework equal to zero. Lean-Kanban approach presented a slowly increasing median value, together with an strong increase of total time variations. The Scrum approach, instead, presented a similar behavior, but centered around higher median values of project duration.

Regarding the total project duration varying the percentage of rework, while keeping the featuresâ effort standard deviation equal to zero, the Lean-Kanban, showed a strong increase both in the median and the variations as rework percentage increases. For Scrum, the increase in median effort and in volatility is even bigger. We note also a strong leap in the first step from no rework to 10% rework, that can be explained perhaps with the fact that even a small rework percentage is able to increase the number of features that are moved from an iteration to the next, thus altering the whole project schedule. If we considered the total project duration varying both the percentage of rework and the featuresâ effort standard deviation, we could observe that these figures substantially confirms the previous obtained results.

In the case with both the highest rework percentage and effort variation, we could see the highest median and the highest variation for both Lean-Kanban and Scrum cases. As regards lead and cycle times, we will focus only on cycle time, because in the reported case all features enter the system at time zero, so lead times are not very significant. We use the median, that is more significant than the mean for this analysis, and for the sake of simplicity we show only a subset of the computed cases.

Regarding the risk assessment (case study 4) we showed the result that come from the comparison between two methods (Kanban and Scrum) in the no-Risk case. The Lean-Kanban approach regarding a superior performance in all parameters – shorter duration, lead and cycle times, and lower variations of these times. In particular, cycle times, representing how long it takes to implement a feature, from the start of the work on it to its release, are 2-3 times shorter, and much stabler. We, also cosidered, the total project duration varying the standard deviation of features' effort, keeping their average at 2 man-days,

and keeping the percentage of rework equal to zero. Lean- Kanban, presented an increasing median value, together with an strong increase of total time variations. The Scrum case, instead, shows a similar behavior, but centered around higher median values of project duration. Our results show that Lean-Kanban is consistently more efficient than Scrum as regards project length. This was expected, because on one hand Lean-Kanban has no waste time due to sprint planning meetings at each iteration, and on the other hand its continuous flow fully employs all developers in all activities, but for short transients at the beginning and at the end of the project. Conversely, each Scrum iteration entails such transients. One might argue that such waste of time and effort in Scrum is not realistic, because also Kanban needs meetings, and because during sprints team members specialized in activities such as analysis and testing will anyhow find something to do.

Chapter 9

Threats to validity

In this thesis, we presented a method to model and simulate software development and maintenance processes, and some results of its application on different empirical cases. In this section, we discuss what it consider to be the most important threats to validity that should be taken into consideration when considering the presented results, and applying the proposed method. We recall that there are four main types of threats to validity, namely construct validity, internal validity, external validity.

9.1 Threats to internal validity

Internal validity:

An experiment is said to possess internal validity if it properly demonstrates a causal relation between two variables (example fig. 9.1), usually, the treatment and the outcome of the experiment [56] [57]. Internal validity is not relevant for this study as we are not seeking to establish the casual relationship between variables in a statistical manner. In other words, there may be unknown, hidden factors that may affect the results.

In our case, in some experimental cases 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 get 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 development and maintenance processes. Furthermore, our model does not consider the interactions among developers, which may have an impact on the maintenance and development efforts.

9.2 Threats to external validity

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 [56] [57]. In this study we only run the simulation model on some 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 fact an important aspect to be taken into consideration is that data analyzed are synthetically produced and refer to just one simplified test case. Also the fact that, in some case studies all features were available at the beginning of the simulated project, and that no more feature was added, limits the generalization 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 request is often small, thus making the simulation and statistical analysis inappropriate.

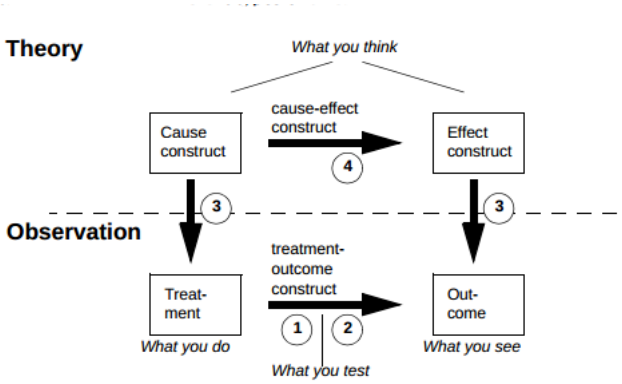

Figure 9.1: *The interpretation of the different types of threats*

9.3 Threats to construct validity

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, therefore, is whether the sampling particulars of a study can be defended as measures of general constructs [56]. In our case, to show construct validity, the appropriateness of our approach, focused on issues, activities and developers, to model the actual software development process, at least within reasonable accuracy, should be checked. We note that decomposing software development in small *features* to be implemented independently from each others is typical of maintenance (where features are called *issues*). This is also a practice common to all agile methods [57].

Also, the fact that feature implementation or issue resolution is made through a sequence of activities performed by a team of developers, skilled in one of more of these activities, is

quite straightforward. So, to a first approximation, our model fits well with *real* software development. What is left out from the model are some kinds of interactions among developers, i.e. the fact that sometime a developer must stop working to ask for advice to another developer. 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, like, 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.

Also, we don't explicitly account for the risk that an entire group of features are badly implemented, and need to be reworked later at great project's expenses. While these characteristics are common to many projects, what we proposed is a model able to represent the main aspects of agile software development, without trying to model every possible aspect.

We showed in previous Sections (Case Study 1 7.2, Case Study 2 7.3 Case Study 3 7.4 and Case Study 4 7.5) that this model is able to replicate quite well the empirical characteristics of the real maintenance and development projects studied. A threat to the validity of this replication is that of overfitting empirical data might have been well reproduced owing to the very many parameters used to tune the model.

Regarding the Microsoft case (Case study 2 7.3 and second part of Case study 3 7.4), we tuned just a few parameters (average effort of the issues and penalty factor p) to reproduce the overall throughput of the team, and we got cycle time statistics very close to real data, as reported in Tables 7.2 and 7.3. We did not optimize any parameter to get these statistics right.

Regarding the Chinese firm case (case study 3 7.4) we just tuned the same parameters (and the team size in correspondence to the throughput variations at 400 and 800 days), getting a very good agreement throughout all relevant time intervals. So, overfitting should be ruled out. Regarding the risk assessment (case study 4) , the main threat to construct validity is whether our models of Scrum and Lean-Kanban processes are accurate enough to be realistic. In other words, are features, activities and developers, as modeled by us, enough to get plausible results? Other studies on empirical data seem to answer favourably to this question [57] [51] [87], but more research is clearly needed. Another issue related to construct validity are the characteristics of feature effort variations, and of the need of rework. While these characteristics are common to many projects, the exact distribution of effort variations, and the way rework is performed might be improved. Moreover, there are other possible risk factors, such as inaccurate requirements and team-related issues, such as resignation of developers, introduction of new developers, intra-team dependencies, and so on.

Chapter 10

Conclusion and future work

This section is dedicated to discuss the main aspects, regarding the simulation model and its applications. We recall some important elements we faced in this work thesis highlighting the obtained results and useful purposes of simulative approach.

We presented a process simulation model developed for assessing the effectiveness of the WIP-limited approach, and to visualize the flow of work and the organization of work. The model has been designed and implemented using a full object-oriented approach, and is easily extensible to accommodate more detailed and realistic representations of the actual software development process. The simulation outputs can be elaborated, yielding all kinds of diagrams and statistical information in order to analyze every aspect of what actually happened during the simulation. We showed simulation results on WIP-limited and unlimited developments, varying the skills of the developers involved, showing substantial differences in the resulting CFDs. In this thesis we have presented a simulation model customized, also, for software maintenance process, that was used for assessing the effectiveness of agile and lean approaches described in [3] to a real case study, in which a maintenance team experimented the transition from a general, 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 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.

Furthermore we used the simulator to optimize, through exhaustive search, the WIP limits of the activities of the process. We analyzed the repeatability and robustness of the results, showing that the simulator can be very useful under this respect.

We analyzed some processes regarding development and maintenance processes following the Lean-Kanban principles and practices the other belonging to the lean approach with respect to the risk management and in order to choose the one that better addresses the related issues and making also a comparison with other general processes in order to show how lean-Kanban approach does improve it.

Further studies extended the simulator toward more realistic settings. Among others, we worked on the following extensions:

- Non-instant availability of collaborators. Could be a specialist required to perform a task such as *"code security review"* or a business person, subject matter expert re-

quired to disambiguate the requirements. Modeling non-instant availability of collaborators with different delay parameters would be useful.

- Special cause variation – work blocked due to external circumstances such as a test environment failure, lack of availability of a collaborator due to special cause reason (illness, vacation, business trip).

- Specialization of the workforce - for example, the design activity might be made up of various optional sub-activities, e.g. user interface design, database design, Web services design, etc. Some of these sub-activities may require a specialist, or one from a smaller pool of suitably talented people.

We could model the mix of work items that require more or less specialization and model the number of sub-activities that have one or very few capable people, and simulate the effect. This modification would imply to explicitly model the decomposition of features in tasks. Management of high-priority features, and the effect of the team choosing to break the WIP limit from time-to-time. Explicit management of software product quality and bugs, with the need to rework at least partially some feature to fix its bugs.

We used, as input data, a stream of requests synthetically generated, with the same statistical properties of real requests. The simulator was able to fully reproduce the statistics of empirical results for the original process, both in terms of throughput and cycle times. The proposed approach to model and simulate a software process, using an agent-based, fully object-oriented model, even in this case demonstrated very effective. It allowed us to model different 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.

By simulation, we showed also 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.

Another result of our study was a comparison between Scrum and Lean-Kanban processes, whose results confirm what is being reported in this study. The comparison showed that Scrum is less efficient and more prone to risk than Kanban as regards productivity and cycle times. This result is not unexpected, given the assumptions of our model. Of course, Scrum has other advantages over Kanban, and both methods can also be merged, trying to obtain the best of the two, as in Scrumban. However, these results can be useful, together with other results and considerations, to a manager with the task to assess the most suited method to adopt. In the future, we will improve our risk assessment method, evaluating it on many other case studies, and also exploring the optimal parameter settings that can minimize the overall risk.

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 could also analyze and model the human and team interactions factors that could affect

a project team's maintenance performance. A substantial improvement to our model we are considering is to scale the model from a single team to multiple teams involved in multiple projects. This would greatly improve the utility of the tool for large organizations.

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.

In this thesis we tried to also answer the typical questions regarding the generalization of the obtained results, typical customization needed in order to reproduce, calibrate all aspects of the process, replications of input data (equivalent to statistic values and results). In the future we will gather data from next Software Factory projects and from industrial projects that will allow to validate more the reliability of the simulation model that is suitable to reproduce different processes.

In particular we aim to conclude the section of research including a simulative study that we are using to better understand software processes for distributed development in the cloud environment, in the context of a Software Factory network. Based on the observation of a real project with sites in Finland and Spain and a Scrumban-like process, an existing simulator has been adapted to reflect the Scrumban process. The goal of creating the simulation model was to better understand the distributed software process (with the cloud environment as context). The goal of the simulation model itself is to support decisions for planning such kind of projects. Considering the threats of validity of the study, the accuracy and reliability of the simulation model could be shown and the simulation model implementation allows for deriving hypothesis on the impact of distribution on parameters such as throughput.

Overall, we believe that the presented work demonstrated that our event-driven and agent-based approach is very effective for modelling 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. At the same time, the study provided important elements to evaluate the effectiveness of the simulator that could help increase the understanding of relationships between important variables such as size and WIP, and can be used as a means to support decision making and to forecast and manage the risk in software process development.

Bibliography

[1] Poppendieck, M., and Poppendieck, T.,: Lean software development: An agile toolkit. Boston, Massachusetts, USA: Addison Wesley, 2003. [cited at p. 1, 7]

[2] Womack, J.P., Jones, D.T., and Roos, D. :(1991), The Machine That Changed the World: The Story of Lean Production, HarperBusiness. [cited at p. 2, 7]

[3] Anderson, D.J.: Kanban: Successful Evolutionary Change for Your Technology Business, Blue Hole Press, 2010. [cited at p. 1, 8, 13, 27, 56, 58, 111]

[4] Kellner, M., Madachy R.J., and Raffo, M.: Software process simulation modeling: Why? What? How. Journal of Systems and Software, vol. 45, 1999, pp. 91-105. [cited at p. 9, 33, 35, 43, 44]

[5] Zhang, H., Kitchenham, B., and Pfahl, D.: Reflections on 10 years of software process simulation modeling: a systematic review, Proceedings of the International Conference on Software Process, ICSP 2008, LNCS vol. 5007, Springer, 2008, pp. 345-356. [cited at p. 9, 43]

[6] Martins, K., Lewandrowski, U.: Inventory safety stocks of Kanban control systems. Production Planning and Control, vol. 10, 1999, pp. 520-529. [cited at p. 1]

[7] Huang, P.Y, Rees, L.P. and Taylor BW.: A simulation analysis of the Japanese just-in-time technique (with kanbans) for a multiline, multistage production system. Decision Sciences, vol. 14, 1983 pp. 326-344 [cited at p. 7, 26]

[8] Hurrion, R.D.: An example of simulation optimization using a neural network metamodel: finding the optimum number of kanbans in a manufacturing system. Journal of the Operational Research Society, vol. 48, 1997, pp. 1105-1112. [cited at p. 8, 26]

[9] Köchel, P., and Nieländer, U.: Kanban Optimization by Simulation and Evolution. Production Planning & Control, Vol. 13, 2002, pp. 725â734. [cited at p. 8, 26]

[10] Hao, Q., and Shen, W.,: Implementing a hybrid simulation model for a Kanban-based material handling system. Robotics and Computer-Integrated Manufacturing, vol. 24, 2008, pp. 635-646. [cited at p. 8, 26]

[11] 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. [cited at p. 9, 26]

[12] 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. [cited at p. 9, 26]

[13] Ladas, C.: Kanban simulation, online at:leansoftwareengineering.com/2008/11/20/kanban/-simulation/, December 2010. [cited at p. 8, 26]

[14] Bowden, R.O., Hall,J.D. :Simulation optimization research and development, Proc. Winter Simulation Conference (WSC'98), 1998, pp.1693-1698. [cited at p. 2, 8, 55]

[15] Ohno, T.: Just-In-Time for Today and Tomorrow, Productivity Press, 1988. [cited at p. 7]

[16] Cockburn, A.: Crystal Clear: A Human-Powered Methodology for Small Teams, Addison Wesley, 2004. [cited at p. 7, 56]

[17] Ladas, C.: Scrumban. Modus Cooperandi Press, Seattle, WA, USA, 2008. [cited at p. 1, 7, 8, 56]

[18] Kniberg, H., and Skarin, M.: Kanban and Scrum making the most of both, C4Media Inc, 2010. [cited at p. 8, 56]

[19] Craig, D.C., :Extensible hierarchical object-oriented logic simulation with an adaptable graphical user interface. Master of science, School of Graduate Studies, Department of Computer Science, Memorial University of Newfoundland, 1996. [cited at p. 38]

[20] Park, S., Kim H., Kang D., Bae D.H., :Developing a software process simulation model using SPEM and analytical models, Volume 4, Number 3-4 , pp 223 â 236, 2008. [cited at p. -]

[21] Donzelli, P., Iazeolla, G., :Hybrid Simulation Modelling of the Software Process, Journal of Systems and Software, Volume 59, Issue 3, pp 227-235, 2001. [cited at p. -]

[22] Donzelli, P., and Iazeolla, G., :Using Process Models to test Process Assumptions within the SEL Recommended Development Approach, 25th NASA Software Engineering Workshop, NASA-GSFC, Greenbelt, MD, 2000. [cited at p. 44]

[23] Choi, K., Bae, D. and Kim, T., :An approach to a hybrid software process simulation using DEVS formalism. Software Process: Improvement and Practice, Vol. 11, No. 4, pp.373â383, 2006. [cited at p. -]

[24] Martin, R., Raffo, D., :A model of the software development process using both continuous and discrete models., Software Process: Improvement and Practice, 5, pp. 147â157, 2000. [cited at p. 39, 40, 42, 44]

[25] Schriber, T.J., Brunner, D.T., :Inside discrete-event simulation software: how it works and why it matters. In WSC â99: Proceedings of the 31st conference on Winter simulation, pages 72â80. ACM Press, 1999. [cited at p. 41]

[26] Münch, J., Armbrust, O., Kowalczyk, M., Soto, M., :Software Process Definition and Management The Fraunhofer IESE Series on Software and Systems Engineering [cited at p. -]

[27] Kreutzer, W., :System Simulation - Programming Styles and Languages. Addison Wesley, Reading (U.S.A.), 1986. [cited at p. -]

[28] Fishman, G.S., :Discrete-Event Simulation: Modeling, Programming, and Analysis. Springer Series in Operations Research and Financial Engineering. Springer-Verlag, Berlin, 2001. [cited at p. 41]

[29] R., Martin, D., Raffo,: Application of a hybrid process simulation model to a software development, Journal of Systems and Software, Volume 59, Number 3, 15 December 2001 , pp. 237-246(10) [cited at p. 11, 44]

[30] Rus I., Collofello J., Lakey P., :Software process simulation for reliability management., Journal of Systems and Software , 46(2â3), pp. 173â182, 1999. [cited at p. 43]

[31] Lakey P., :A hybrid software process simulation model for project management. In Proceedings of the International Workshop on Software Process Modeling and Simulation (ProSim â03), Portland, OR, 2003. [cited at p. 43]

[32] Wallace, L., Keil, M., Rai, A.: How software project risk affects project performance: An investigation of the dimensions of risk and an exploratory model, Decision Sciences, 35, 289-321 (2004). [cited at p. 73]

[33] Forrester, J.W. :Industrial Dynamics. Cambridge MA: Productivity Press, 1961 [cited at p. 39]

[34] Jones, C. :The Economics of Software Maintenance in the Twenty First Century, Available online at: www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf, February 14, 2006. [cited at p. 1, 3]

[35] The system Dynamics in Education Project MIT Road maps: A guide to learning system dynamics. Published on: http://sysdyn.clexchange.org/road-maps/home.html, 2000 [cited at p. 3, 39]

[36] Fishman, G.S., :Discrete event simulation: Modeling, Programming, and Analysis. Springer Series in Operations Research and Financial Engineering. Springer-Verlag, Berlin, 2001 [cited at p. 40, 71]

[37] Ohno T., Mito S., Schmelzeis J., :Just-In-Time for Today and Tomorrow, Productivity Press, Cambridge, MA, 1988. [cited at p. 10]

[38] Poppendieck, M., Poppendieck, T. :Implementing Lean Software Development From Concept to Cash. Addison Wesley, Boston, 2006. [cited at p. 1, 10]

[39] Poppendieck, M., Cusumano, M.A. :Lean Software Development: A Tutorial, IEEE Software, vol. 29, pages 26-32, 2012. [cited at p. 9]

[40] Petersen, K., Wohlin, C. :Measuring the Flow in Lean Software Development. Software Practice and Experience, vol. 41, 975-996, 2011. [cited at p. 9]

[41] 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, pages 305-314, Las Vegas, NE, USA, 2011, IEEE Computer Society Press. [cited at p. 2]

[42] 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, vol. 85, pages 1287â1299, 2012. [cited at p. 2, 10]

[43] Sjøberg, D.I.K, Johnsen. A., Solberg, J. :Quantifying the Effect of Using Kanban versus Scrum: A Case Study, IEEE Software, vol. 29, pages 47-53, 2012. [cited at p. -]

[44] Abdel-Hamid, T., Madnick, S. :Software Project Dynamics: An Integrated Approach, Prentice-Hall, Upper Saddle River, NJ, 1991. [cited at p. 2, 39]

[45] Madachy, RJ., :Software Process Dynamics,Wiley-IEEE Press, Chichester, UK, 2008. [cited at p. -]

[46] 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), Reston, Virginia, USA, October 10-11, 1994, IEEE CS Press, 99-110. [cited at p. 8]

[47] 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, 240â249, Los Alamitos, California, USA, 2005. IEEE Computer Society Press. [cited at p. 3]

[48] 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 30, 1 (2004), 43-58. [cited at p. 9]

[49] Lin, CT., Huang, CY., :Staffing Level and Cost Analyses for Software Debugging Activities Through Rate-Based Simulation Approaches. TR, Dec. 2009, 711-724. [cited at p. 10]

[50] Anderson, DJ., Concas, G., Lunesu, M.I., Marchesi, M. :Studying Lean-Kanban Approach Using Software Process Simulation. Agile Processes in Software Engineering and Extreme Programming 12th International Conference, XP 2011, Madrid, Spain, May 10-13, 2011, Springer LNBIP vol. 77, 12-26. [cited at p. 9, 26, 74, 75]

[51] Anderson, DJ, Concas, G., Lunesu, M.I, 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 12th International Conference, XP 2012, Malmoe, Sweden, May 21-25, 2012, Springer LNBIP vol. 111, 123-137. [cited at p. 8, 9, 26, 74, 109]

[52] 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), March 19-22, 2012, St. Louis, MO, Procedia Computer Science, vol. 8, pages 309â314. [cited at p. 9, 10, 26]

[53] 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, pp. 23-27. [cited at p. 9, 10, 26]

[54] Curtis, B. :Substantiating Programmer Variability. Proceedings of the IEEE, vol. 69, no. 7, 1981. [cited at p. 85, 88]

[55] Wolverton, RW. :The Cost of Developing Large-Scale Software, IEEE Trans. on Computers, vol 23, 1975, 615-636. [cited at p. 58]

[56] Shadish, W., Cook, T., Campbell, D. :Experimental and Quasi-Experimental Designs for Generalized Causal Inference. Houghton- Mifflin, Boston, 2002. [cited at p. 107, 108]

[57] Highsmith, J. :What is Agile Software Development? CrossTalk, The Journal of Defense Software Engineering, October 2002, 4-9. [cited at p. 107, 108, 109]

[58] Albrecht, A.J. :Measuring application development productivity. Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium, 83â92, Monterey, California, October 1979, IBM Corporation. [cited at p. -]

[59] 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, 3 June 2012, Zurich, Switzerland, 14-20. DOI 10.1109/WETSoM.2012.6226986. [cited at p. -]

[60] Humphrey, W.S., :Introduction to the Team Software Process, Addison Wesley, 1999. [cited at p. 8, 70]

[61] 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. [cited at p. 8]

[62] Robinson, S.: Simulation – The practice of model development and use. Wiley, Chichester, UK, 2004. [cited at p. 61]

[63] Schwaber, K., Beedle, M. :Agile software development with Scrum. Prentice Hall (2002). [cited at p. 8]

[64] 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. [cited at p. 61]

[65] Version One.: State of Agile Survey 2010. Online at www.versionone.com. [cited at p. 8, 60]

[66] Maurer, F., Martel, S. :On the productivity of agile software practices: An industrial case study. Retrieved September 20, 2004. [cited at p. 96]

[67] 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. IFIP Central and East European Conference on Software Engineering Techniques, 2007. [cited at p. -]

[68] Abrahamsson,cP., Warsta, J., Siponen, M.T., Ronkainen,J., :New directions on agile methods: a comparative analysis, in: Proceedings of the 25th International Conference on Software Engineering (ICSEâ03), IEEE Press, 2003. [cited at p. -]

[69] Dyb, T., and Dingsyr, T., :Empirical Studies of Agile Software Development: A Systematic Review, Information and Softkanbaware Technology, vol. 50, nos. 9/10, pp. 833-859, 2008. [cited at p. -]

[70] Banks, C.M., Sokolowski, J.A., :Principles of Modeling and Simulation: A Multidisciplinary Approach, Edited by Published by John Wiley & Sons, Inc., Hoboken, New Jersey. [cited at p. 31]

[71] Zhang, H., Jeffery, R., Houston, D., Huang, L., Zhu, L., :Impact of Process Simulation on Software Practice : An Initial Report, ICSE â11 Proceeding of the 33rd International Conference on Software Engineering, pp. 1046â1056, ACM, NY (USA) [cited at p. 32, 34]

[72] A., Lamersdorf, J., Münch, A., FernÃ¡ndez- del Viso Torre, C., Rebate Sánchez, M., Heinz and D. Rombach,: A rule-based model for customized risk identification and evaluation of task assignment alternatives in distributed software development projects- ICGSE '10 Proceedings of the 2010 5th IEEE International Conference on Global Software Engineering table of contents 209-218 Publisher IEEE Computer Society Washington, DC, USA Â©2010 ISBN: 978-0-7695-4122-8 [cited at p. 9]

[73] Cavrak,I., Orlic, M., Crnkovic, I.,:Collaboration patterns in distributed software development projects-34th International Conference on Software Engineering (ICSE), 2012 , Page(s):1235-1244 [cited at p. 10]

[74] Alberts, C.J, Dorofee, A.J.: Risk Management Framework . Pittsburgh: Carnegie Mellon, Software Engineering Institute. Techn. Report CMU/SEI-2010-TR-017 (2010). [cited at p. -]

[75] Chittister, C., Haimes, Y.: Assessment and Management of Soft-ware Technical Risk, IEEE Transactions on Systems, Man, and Cybernetics 24, 187-202 (1994). [cited at p. -]

[76] Lowrance, William W.: Of Acceptable Risk: Science and the Determination of Safety. Los Altos, Ca, William Kaufmann (1976). [cited at p. -]

[77] Rutherford, K., Shannon, P., Judson, C., and Kidd, N. 2010. :From Chaos to Kanban, via Scrum,Proceedings of the 11th International Conference on Agile Software Development, XP2010, Trondheim, Norway: Springer Verlag, pp. 344-352 [cited at p. 101]

[78] Charette, R.N., :Why software fails, IEEE Spectrum, 2005. 42(9): pp. 42-49. [cited at p. 8, 101]

[79] Boehm, B.W., :Software risk management: principles and practices, IEEE Software, 1991. 8(1): pp. 32-41. [cited at p. 8]

[80] Dedolph, F.M., :The Neglected Management Activity: Software Risk Management, Bell Labs Technical Journal, 2003. 8(3): p. 91-95. [cited at p. 9]

[81] Pandian, C.R., :Applied Software Risk Management: A Guide for Software Project Managers, Auerbach Publications, 2006. [cited at p. 9]

[82] Liu, D., Wang, Q., Xiao, J.: The role of software process simulation modeling in software risk management: A systematic review. In: Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM 2009), Lask Buena Vista, FL, October 2009, pp. 302â311. IEEE Computer Society, Los Alamitos (2009) [cited at p. 9]

[83] Nikitina, N., Kajko-Mattsson, M., :Developer-Driven Big Bang Process Transition from Scrum to Kanban, Proc. 2011 International Conference on on Software and Systems Process (ICSSP 2011), ACM, 2011. [cited at p. 10, 101]

[84] Cao, L., Ramesh, B., and Abdel-ÂHamid, T. :Modeling Dynamics in Agile Software Development. ACM Transactions on Management, 1 (2010). [cited at p. 9]

[85] Abdel-Hamid, T.K., Madnick, S.E.: Software Project Dynamics: An Integrated Approach. Prentice Hall, Englewood Cliffs (1991) [cited at p. -]

[86] Cocco, L., Mannaro,K., Concas, G., and Marchesi, M., :Simulating Kanban and Scrum vs.Waterfall with System Dynamics. Agile Processes in Software Engineering and Extreme Programming Lecture Notes in Business Information Processing, 2011, Volume 77, Part 1, 117-131,. [cited at p. 9]

[87] Concas, G., Lunesu, M.I., Marchesi, M., Zhang, H.: Simulation of Software Maintenance Process, with and without a Work-In-Process Limit. submitted for publication (2013). [cited at p. 109]

List of Publications Related to the Thesis

Published papers

Conference papers

- Anderson, D.J., Concas,G., Lunesu, M. I., and Marchesi, M.,: *Studying Lean-Kanban Approach Using Software Process Simulation.* in *Agile Processes in Software Engineering and Extreme Programming 12th International Conference*, XP 2011, Madrid, Spain, May 10-13, 2011, Springer LNBIP vol. 77, pp. 12-26.

- Anderson, D.J., Concas,G., Lunesu, M. I., Marchesi, M., and Zhang, H.,: *A comparative study of Scrum and Kanban approaches on a real case study using simulation.* in *Agile Processes in Software Engineering and Extreme Programming 12th International Conference*, XP 2012, Malmoe, Sweden, May 21-25, 2012, Springer LNBIP vol. 111, pp. 123-137.

- Concas, G., Lunesu, M.I., Marchesi, M., and Zhang, H., *Simulation of Software Maintenance Process, with and without a Work-In-Process Limit* Published in Journal of Software: Evolution and Process

Submitted papers

- Concas, G., Lunesu, M.I., and Marchesi, M. *Assessing the risk of software development in agile methodologies using simulation* Submitted

Not yet Submitted papers

Lunesu, M.I., Münch, J., and Marchesi, M. Simulation of a Distributed Software Development Project in the Cloud

Appendices

Appendix A

Extra Data

This section collects additional content to those presented in Experimental Results chapter.

In this section we presented the analysis made on the data that come from the Chinese firm. This section represent the pre-processing phase needed before the use of data as input of the simulator. After this important phase, the data was used as input of the simulation model in order to perform hundreds of simulation runs useful to assess the simulator and to produce output tha we can see in the Experimental Results section. In this section we show the detailed analysis made as follows.

The dataset consists of 5854 sets of data, each referring to a bug. The dates associated to the bugs vary between 2005 and 2010. For each bug, there are the following 12 data (non always set):

- Issue Id
- Request Type
- Issue Status
- Issue State
- Report Date
- Product Version
- Component
- Submit Peg
- Analysis Time
- Answer Date
- Date Of Change
- Fixed Time
- Verify Date

Table 1: Days of arrival of bugs.

Day	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Bugs reported	1127	1104	1111	1104	782	179	447
State change	1043	1073	1156	990	817	253	522

Some observations about the data:

- The most important subdivision seems between bugs whose *Issue Status* is CLOSED and those whose status is OPEN.
- There are 3839 bugs CLOSED, and 2015 OPEN.
- The *Issue States* of CLOSED bugs can be only three: *Resolved*, *NoFurtherAction* and *Duplicate*.
- The *Issue States* of OPEN bugs can be ten: *NoFurtherAction Postponed Duplicate Suspend NotAccepted Submitted NeedMoreInfo Updated Analyzed Resolved*.
- All bugs have always set their *Report Date* and *Date Of Change*.
- CLOSED bugs have always set their *Answer Date* and *Verify Date*.
- OPEN bugs have often not set their *Answer Date* and *Verify Date*.
- The fields *Analysis Time* and *Fixed Time* are set only sporadically.

The weekdays the bugs were reported or changed state are summarized in Table 1:

Most of the activity took place in the first four days of the week. A reduced activity is present on Friday. The lowest activity is on Saturday – though not absent. On Sunday there is a fair amount of activity. For the analysis and simulation of the process, we might consider a six-day week, in practice merging Saturday and Sunday activities in one day. The proposed models do not consider the lower activity in the last two days of the six-day week, but will average out these week-end effects.

Information coming from the database

Initial information about the bug-fixing team was that is was composed of about 6-7 members, though this number varied. Further information collected from the student:

- There are two teams: the Developers who fix the bugs, and the Testers who submit the bugs and verify that they are fixed.
- The Developers can set the following states: *Resolved*, *NoFurtherAction*, *Analyzed*, *Suspend*, *Duplicate*, *Updated*.
- The testers can set *Submitted*, *Updated*, *NotAccepted*.
- The Answer Date is the date when developer claim a bug is fixed.
- Once a developers claims a bug fix, he will set the bug's states as shown above.

Table 2: Question and Answers about bug fixing process.

Question	Answer
How many developers (full- and part-time) worked in the development team?	The student doesn't know exactly as the project was organized in a global-development style. The team size is also dynamic (with people move in and out) over the years. The student is within the testing group (not the development group) so not sure how the development work is organized...
What was the average time of developers' work needed to fix a bug?	This time should include trouble shooting time, the average time maybe related with the bug priority and difficulty Level. Now it is difficult for me to states the average time.
Why so many bugs took so long (even months) to be verified/closed after fixing?	Some of these bugs could be Reopened (Not Accepted)---Verified many times, the recorded Verified time is the last verification time recorded by the system. For some bugs, after they're fixed by developers, then testers will verify them in the new release version. Some bugs maybe rejected by testers due to they're not fixed well and the issue still existed in the new version. These bugs will be re-fixed by developers, then re-verified by testers.
What is the percentage of bugs that did not pass the final verification phase, and had to be reworked?	The student cannot give a precise estimation. It is however about 20%

- The testers then verify it. If *NotAccepted*, the bug is set to OPEN. Otherwise, the bug is set to CLOSED.
- In average, it took testers 15 minutes to verify a bug fix.
- When a bug is created, the state is Submitted. For some bugs, when additional information comes the bug status is changed to Updated.
- For some bugs, the developers think they are related to certain user-requests and new features, so the developers can create Enhancement for them. Around 30%-40% enhancements are related to bugs.
- For these bugs, the fixing time are usually long.
- In general, bugs with higher Submit Peg will be handled first. Bugs with lower Submit Peg values will be dealt with later.
- All dates in the EXCEL file were generated automatically by the system when an action was taken by either developers or testers. The student was a tester and doesn't know much about the organizational structure of the development team. Some more questions were submitted to the student, summarized with their answers in Table 2.

Issues and their states

A possible state diagram of the *Issue States* and their transitions is shown in Fig. 1. Note that *NotAccepted* state is marked as *final*, though all 39 bugs in this state are still marked as OPEN. Some of these bugs have a reporting date of 2005 and 2006, so it is argued that this state will never change further. In the diagram, it is assumed that a bug is always analyzed (except for *NotAccepted* bugs) and, if it is suspended or if more information is needed, it is then analyzed again. In the followings, we will start considering only CLOSED bugs, because they are more complete and suitable to be studied.

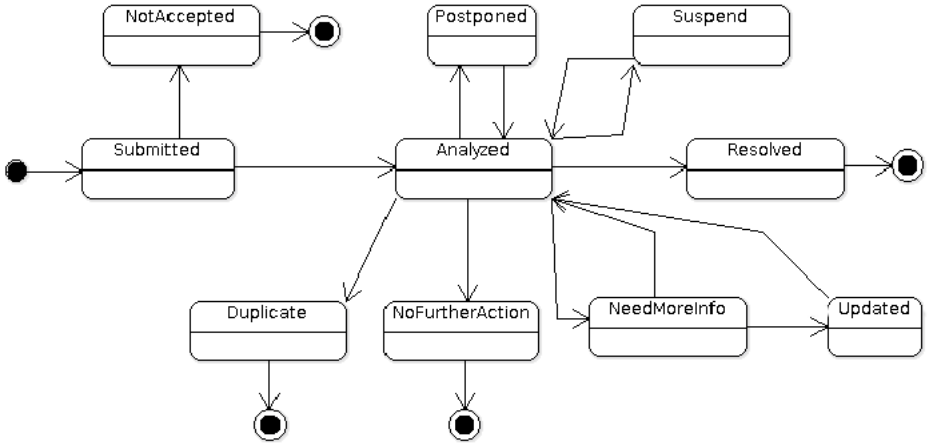

Fig 1. A possible state diagram of bug fixing

Table 3: Statistics of the times needed to fix bugs as a funcion of their final state.

State	Quantity	Min	1st Qu.	Median	Mean	3rd Qu.	Max
Res.	Answ. time	0	6	24	90.31	90	1694
Res.	Answ. to verify	0	9	20	46.37	45	627
Res.	Ver. to change	0	0	0	4.176	0	470
NFA	Answ. time	0	8	47.5	170.9	196	1854
NFA	Answ. to verify	0	5	18	52.89	50	564
Dupl.	Answ. time	0	1	9	69.01	75.25	1093
Dupl.	Answ. to verify	0	15	34	72.62	93	559

CLOSED Bugs Analysis

CLOSED bugs are 3839. Among them, Duplicate are 456, *NoFurtherAction* are 898, and Resolved are the remaining 2485. The followings table reports basic statistics of the times (in days) elapsed from the report date to the answer date (Answer Time), and from the answer date to the verify date (Answer to Verify). Note that the verify dates and the change dates are almost always the same, so the time from Verify to Change is almost always zero. Table reports some statistics on this time only for Resolved bugs.

Defects vs. Enhancements

In the analyzed data, there are 5038 defects and 816 enhancement requests. Table 4 reports some data about OPEN and CLOSED defects and enhancements.

Note that the relative percentage of enhancements with respect to defects is about 10% for CLOSED issues, and becomes 31% for open issues. This means that enhancements tend to stay open for a longer time – this is probably due to two causes: the defect correction is generally more urgent than enhancement upgrading, and enhancements need usually more work than defects. The average and standard deviation of the days to fix defects for closed issues is reported in Table 5:

Table 4: Statistics about bugs, as a function of their type and final status.

	CLOSED		OPEN		TOTAL
Type	Nr.	Percent.	Nr.	Percent.	Nr.
Defect	3497	69.4%	1541	30.6%	5038
Enhancement	342	41.9%	474	58.1%	816
%Enh. / Defects		9.78%		30.8%	16.2%
%Enh. / All		5.84%		8.1%	13.9%

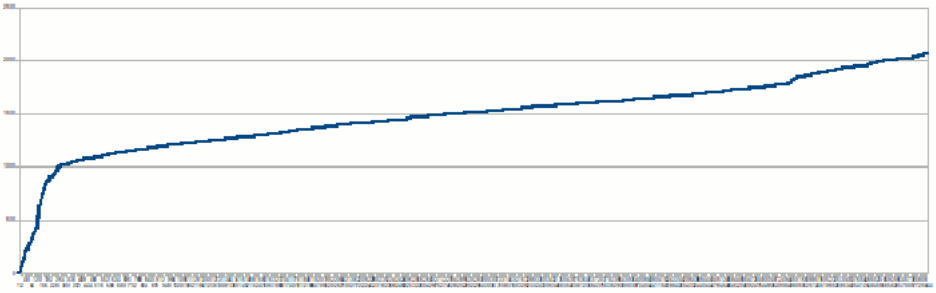

Fig 2: View of all arrival times (days from 1/2/2005), ordered.

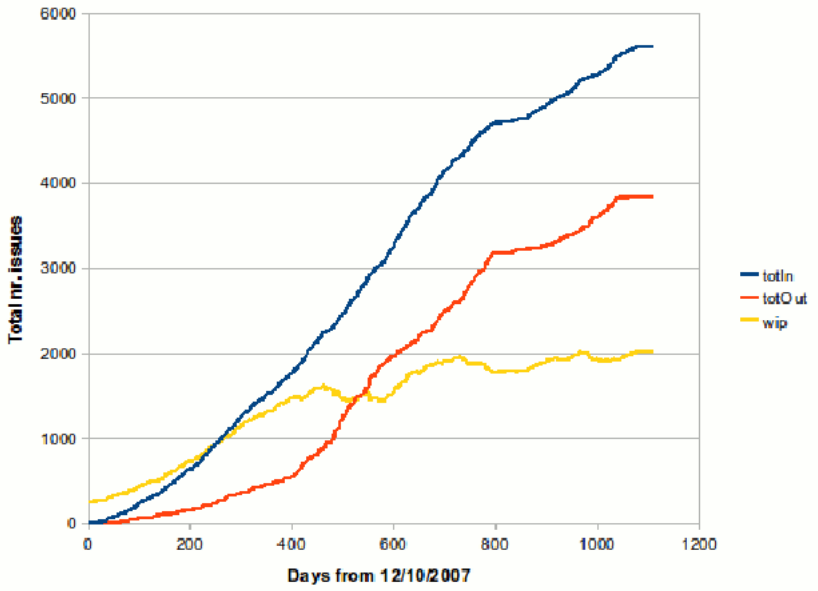

Fig 3: Cumulative nr. of issues in the system.

Times of arrival and issues in and out

In Fig. 2 we show the data representing the day of arrival of all issues, with respect to 1/2/2005, that is the report date of the first issue recorded. All the 5854 data are ordered by day, and are reported along x axis; their day is reported along y axis. Note that in the first 1000 days (until roughly 1/11/2007) only 251 issues arrived – about one in three days. In the period from 1/11/2007 to 1/11/2010 there were about 5600 issues, with an average of about 5 issues per

Table 5: Average and standard deviation of times to close bugs and enhancements.

Type	Mean	St. Dev.
Defect	152.9	233.0
Enhancement	238.6	220.1

Table 6: Statistics about bug arrival and fixing time in the three periods with different bug input pace.

Period (days)	New bugs / day	New b/d (6 days week)	Fixed bugs / days	Fixed b/d (6 days week)	Team size (average)	Avg. time to fix a bug (days)
1 (1-400)	6	5.14	1.5	1.286	2	1.556
2 (401-800)	6	5.14	6.5	5.571	8	1.436
3 (800-1030)	3	2.57	2.5	2.143	3	1.400

day, at a rate fairly constant, with a slight increase in the last year. In fact, in the dataset there is no item whose Date of Change is before 12/12/2007. This means that requests closed before that date are not reported in the dataset, and explains the strange behavior shown in Fig. 2. The 251 issues âarrivedâ before 12/10/2007 are therefore the issues still under work at that date. Fig. 3 shows the total nr. of issues entered into the system, exited from it, and whose work is in progress as a function of the number of days from 12/10/2007, until 1/11/2010.

As you can see, the team devoted to fix bugs was not able to keep up with the pace of arrival of bugs for about two years (more than 700 days), until the number of issues waiting to be solved reached the number of about 2000. Then, the team 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, that after day 800 changed from about 6 issues/day to about 3 issues/day.

Consequently, we should consider as a starting date for our analysis the date of 12/10/2007, keeping the issues with report date before this date as initial state of the system. In this way, the dataset becomes balanced and unbiased.

Statistics about issue management

When the issue management team cannot keep up the pace of input issues, the size of work in process (WIP) increases, and the number of issues completed (out issues) depends only on the team's capacity, and not on the issues in input. From Fig. 3, it is patent that the number of issues completed follows three different patterns: 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. In the last days, both arriving and closed issues slow down. This looks related to the end of data collection rather than to actual events. We will not consider therefore variations after day 1030, that is after 7/8/2010.

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

Consequently, we assume that such change was due to a change in the team size. The

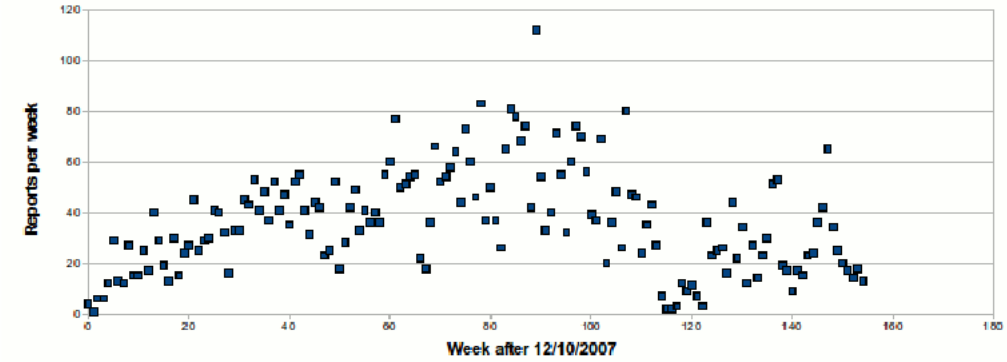

Fig 4: Nr. of bug reports per week. Weeks are progressively numbered from 12/10/2007 to 30/9/2010.

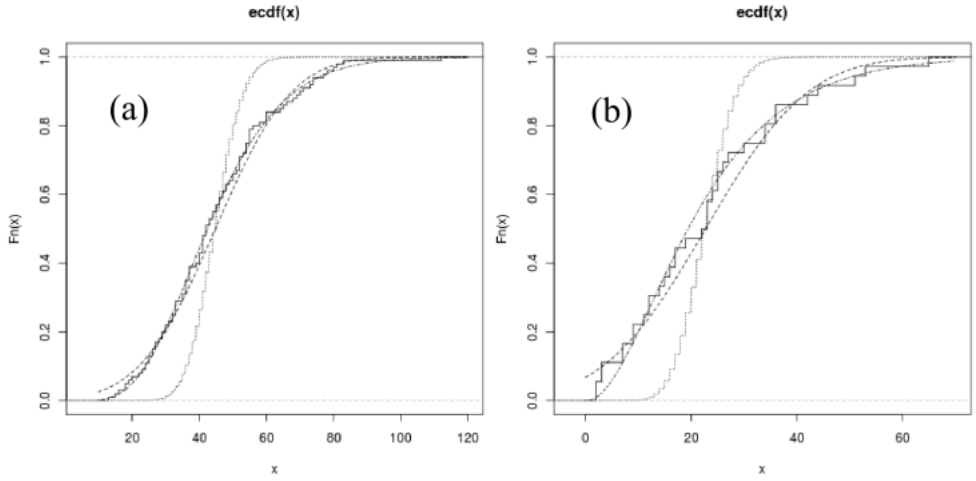

Fig 5: CDF of arrival times (solid line) in weeks 15-114 (a), and 115-150 (b), with best fit distributions: Poisson (dotted line), Normal (dashed line), Gamma (dot-dashed line).

only information we have about team size is an initial estimate of about 6-7 people. We assume that this size refers to the period when most issues per day were fixed on average, Table 6 summarizes the hypotheses on team size and its ability to fix bugs. The different average times to fix bugs are needed to fit empirical averages. They represent the average time of actual work needed to fix a bug, including analysis, coding and final testing (which on average takes 15'). These figures include also possible rework.

If needed, the times shown in Table 6 might change varying the average team size in the various periods, but they would be different anyway. The differences might be explained by the fact that, in different periods, the bug-fixing developers were devoted to bug fixing not at 100% of their time. Regarding arrival times of bug report, Fig. 4 shows the number of bugs reported in each week after 12/10/2007. The bugs are cumulated in weeks owing to the irregular arrival pattern during a week.

Fig. 4 shows a short transitory at the beginning, lasting about 14 weeks. Then arrivals are roughly divided in two regimes. The first 800 days (weeks 15-114), and the last ones (weeks 115-150). The former regime (100 weeks) has mean = 44.8 and standard deviation = 17.9. The latter regime (36 weeks) has mean = 22.7 and standard deviation = 15.3. Fig.5 shows the

Table 7: Statistics of times to analyze the issues, in calendar days.

Case	Nr. samples	Mean	St. Dev.	Median	Min.	Max.
With zeros	3895	107.3	207.1	26	0	1854
No zeros	3762	113.8	211.5	32	1	1854

Table 8: Statistics of times to close the issues from analysis, in calendar days.

Case	Nr. samples	Mean	St. Dev.	Median	Min.	Max.
With zeros	3895	55.46	86.55	22	0	871
No zeros	3623	59.63	88.35	25	1	871

Cumulative Distribution Function of the nr. of reports per week in the two regimes, together with their best-fit distributions â Poisson, Normal and Gamma. The Gamma distribution is by far the best fit of empirical data, and should be used for synthetic data generation.

A Kolmogorov-Smirnov test on the best-fit Gamma distribution rejects the null hypotheses in the first regime (weeks 15-114) with p=0.9913, and in the second regime (weeks 115-150) with p=0.8744. Note that applying the K-S test using Gamma parameters computed from the fitted data is not considered a good statistical practice, though. The K-S test applied to the Normal distribution in the same way gives $p = 0.641$ and $p = 0.758$, respectively. The Shapiro-Wilk test for normality rejects the Normality hypothesis with $p = 0.993$ and $p = 0.939$, respectively.

Analysis of transit times of issues

We first examined the distribution of times to perform analysis, that is the number of days spanning from the arrival date to the date of answer of issues. To fit distributions to these data we had to strip the cases in which there is no answer date, and those whose time is zero. Table 7 summarizes these data, for all reported bugs with an answer date.

Since the studied distribution is clearly fat-tailed, we plotted its Complementary CDF in log-log scale, and tried to fit it with Gamma, Lognormal, Weibull and Negative Binomial distributions. The fit was performed only on positive data. Fig. 6 shows the CCDF of the empirical data, and of the best-fitting distributions. Note that Gamma and Lognormal distributions were fitted using best-likelihood parameters (analytical formula using mean and variance of the data), while Weibull and Negative Binomial parameters were found using an optimization procedure (fitdistr() function of MASS package of R).

No tested distribution is able to perfectly follow the data. Overall, the Weibull distribution looks the best for low values of x, and is also not bad in the tail. We then examined the distribution of times to close the bug from analysis, that is the number of days spanning from the date of answer to the date of change of issues. To fit distributions to these data we had again to strip the cases in which there is no answer date, and those whose time is zero. Table 8 summarizes these data, for all reported bugs with an answer date.

We plotted its CCDF in log-log scale, and tried to fit it with Gamma, Lognormal, Weibull and Negative Binomial distributions. The fit was performed only on positive data. Fig. 7 shows the CCDF of the empirical times, and of the best-fitting distributions, as in the previous case. The Weibull and Lognormal distributions look best to fit these data.

Table 9: Statistics of times to close the issues from verification, in calendar days.

Case	Nr. samples	Mean	St. Dev.	Median	Min.	Max.
With zeros	3895	5.662	46.11	0	0	1032
No zeros	133	165.8	189.6	103	1	1032

Eventually, we examined the distribution of times to close the bug from verification, that is the number of days spanning from the date of verification to the date of change of issues. To fit distributions to these data we had again to strip the cases in which there is no answer date, and those whose time is zero. Table 9 summarizes these data, for all reported bugs with an answer date. Note that in only about 3% of cases there is a delay between verification date and change date. When this is the case, however, this delay tends to be quite long, with a mean of 166 days and a median of 103 days.

We plotted its CCDF in log-log scale, and tried again to fit it with Gamma, Lognormal, Weibull and Negative Binomial distributions. The fit was performed only on positive data. Fig. 8 shows the CCDF of the empirical times, and of the best-fitting distributions. In this case, all four distributions are able to fit the curve very well, with the exception perhaps of the Gamma distribution.

Sub Analysis of transit times as a function of priority and state

We computed basic statistics on the total completion times of CLOSED issues, in function of their priority. We remember that priorities vary from zero (lowest) to 30 (highest). Fig. 9 shows a plot of the reciprocal of the median completion (or transit) time as a function of the priority. The data points follow very well a straight line, denoting that completion time is roughly inversely proportional to priority. The only outlier are times of issues with priority zero, that are smaller than expected, suggesting that a value o zero means *no priority* rather than the lowest possible priority.

Table 10 shows the basic statistics of completion times of CLOSED issues, for the various possible priorities. As you can see, the data relative to priority zero are closer to the data of priority 15 (average priority value) rather than to the data relative to priority one.

The number of issues per priority class looks not evenly distributed, with a clear preference for higher values, the mode being priority = 21. An accurate analysis of these data, however, is not useful because in the simulations we will use real data, with their actual priorities.

A possible Kanban model for bug maintenance

The sequence of activities performed when a bug-fixing or enhancement request arrives is reported in Fig. 10. The amount of work needed to complete a request varies in a broad range, from a few hours to many days of work. The average arrival rate of requests is shown in Table 6, for each of the three periods with different issue input rate. This arrival rate may assumed to follow a Poisson distribution with average issues per day shown in Table 6. Using a Kanban board approach to manage the process, the activities/columns of the Kanban board might be the following:

1. First Screening: this is not an activity subject to limits, but it represents the choice of the work items to put in the input queue. A small percentage of items is immediately marked as *Duplicate* or *No Further Action*, but these items in practice do not furtherly affect the process. The result of this activity is a column *Input Queue* holding the work items to process.

2. Analysis: the first activity of the process, that analyzes the requests. After analysis, some requests are marked as *Duplicate* or *No Further Action*, and are put in the *Done* column. Other requests may require more information, or be suspended for some reason.

3. Coding: the second activity of the process, when the defects are actually fixed, or the enhancements done.

4. Verification: the third and last activity, when the Fig 10: A flow diagram representing the work made is verified and accepted. In some cases, bug-fixing activities and the information flows between them. the bug can be sent back to the analysis phase, because it did not pass the verification. The test team was actually separated from the development team, but in a Kanban board approach this can be easily accounted for by enforcing the constraint that test team members can work only on Verification, while development team members can work only on Analysis and Coding activities.

5. *Done*: the last column of the board, reporting the work items in CLOSED status.

The work items in this system are the bug-fixing or enhancement requests. Each work item is characterized by:

1. An identifier.

1. A *weight*, expressing the needed amount of work (see later).

1. A priority, which corresponds to the *Submit Peg* values of items, from 30 to 0 in decreasing order of importance.

Since the distribution of times actually needed to close the work on each request is fat-tailed, and looks to follow a Weibull distribution, it is sensible to assume also a Weibull distribution of the working time needed to fix bugs, with average given in Table 6, and standard deviation to be determined by empirical data.

Note that, from Tables 7 e 8, the average total time to analyze a request is 107.3 days (91.0 working days of a 6-days week), while the average time to close a request is 162.8 days (139.5 working days). This means that for most of the time the requests were left unattended and nobody worked on them. This fact can partially attributed to external factors – for instance some requests are in *Suspended*, *Postponed* or *NeedMoreInfo* state –an partially to lack of organization. The Kanban approach should help under this respect.

In the model, requests (work items) will enter either reflecting the dates and priorities of real data, or following a Gamma distribution with average rate that differs in the three periods, and can be found in Table 6. A small percentage will be chose at random and immediately marked as *Duplicate* or *NFA* and put in the *Done*, or CLOSED, status.

From time to time, when the number of items in the *TO DO* column goes under a given threshold, items in the backlog are chosen and put in the *TO DO* column (which has a limit).

This is made accounting for the priority and time already spent in the queue of pending requests. Then, items are pulled to the *Analysis* column and work on them is performed. When the analysis is done, some items are randomly chosen and again immediately marked as *Duplicate* or *NFA* and put in the *Done* column. Other items are chosen at random and put in a *blocked* status, for a random time interval, to model the requests in *Suspended*, *Postponed* or *NeedMoreInfo* state.

The effects of the various policies to manage these blocked items will be object of study. After Analysis, the remaining, non blocked items will be pulled to the *Coding* column, where coding takes place. For the sake of simplicity, we may assume that these items cannot be further blocked. The last activity is Verification, performed by members of the team devoted only to this task (to model the fact that development and testing teams were in fact different teams). When an item is verified, it is put to the DONE column. The possibility that an item can miss the verification and is put back to the Analysis phase should be considered, but in a first implementation of the simulator will be skipped.

A final observation: Applying a structured process like Kanban, while the total number of requests processed in the considered time interval will be the same of existing empirical data, the average and maximum time to process a request can drop dramatically. This would mimic the observation made