Software Process Improvement through the Lean Measurement (SPI-LEAM) Method

Kai Petersen[*,a,b], Claes Wohlin[a]

[a]*School of Computing, Blekinge Institute of Technology, Box 520, SE-372 25, Sweden*
[b]*Ericsson AB, Box 518, SE-371 23*

Abstract

Software process improvement methods help to continuously refine and adjust the software process to improve its performance (e.g., in terms of lead-time, quality of the software product, reduction of change requests, and so forth). Lean software development propagates two important principles that help process improvement, namely identification of waste in the process and considering interactions between the individual parts of the software process from an end-to-end perspective. A large shift of thinking about the own way of working is often required to adopt lean. One of the potential main sources of failure is to try to make a too large shift about the ways of working at once. Therefore, the change to lean has to be done in a continuous and incremental way. In response to this we propose a novel approach to bring together the quality improvement paradigm and lean software development practices, the approach being called Software Process Improvement through

*Corresponding author
Email addresses: kai.petersen@bth.se, kai.petersen@ericsson.com (Kai Petersen), claes.wohlin@bth.se (Claes Wohlin)
URL: http://www.bth.se/besq; www.ericsson.com (Kai Petersen), http://www.bth.se/besq (Claes Wohlin)

the Lean Measurement (SPI-LEAM) Method. The method allows to assess the performance of the development process and take continuous actions to arrive at a more lean software process over time. The method is under implementation in industry and an initial evaluation of the method has been performed.

Key words:
Lean Software Development, Software Process Improvement, Quality Improvement Paradigm

1. Introduction

Software process improvement aims at making the software process more efficient and increasing product quality by continuous assessment and adjustment of the process. For this several process improvement frameworks have been proposed, including the Capability Maturity Model Integration (CMMI) [1] and the Quality Improvement Paradigm (QIP) [2, 3]. These are high level frameworks providing guidance what to do, but not how the actual implementation should look like. The Software Process Improvement through the Lean Measurement (SPI-LEAM) method integrates the software quality improvement paradigm with lean software development principles. That is, it describes a novel way of how to implement lean principles through measurement in order to initiate software process improvements.

The overall goal of lean development is to achieve a continuous and smooth flow of production with maximum flexibility and minimum waste in the process. All activities and work products that do not contribute to the customer value are considered waste. Identifying and removing waste helps to focus more on the value creating activities [4, 5]. The idea of focusing on waste was initially implemented in the automotive domain at Toyota [6] identifying seven types of waste. The types of waste have been translated to software engineering into extra processes, extra features, partially done work (inventory), task switching, waiting, motion, and defects [7]. Partially done work (or inventory) is specifically critical [8]. The reason for inventory being a problem is not that software artifacts take a lot of space in stock, but:

- Inventory hides defects that are thus discovered late in the process [8].
- Time has been spent on artifacts in the inventory (e.g., reviewing of requirements) and due to change in the context the requirements become obsolete and thus the work done on them useless [9].
- Inventory impacts other wastes. For example, a high level of inventory causes waiting times. Formally this is the case in waterfall development as designers have to wait until the whole requirements document has been approved [9]. Long waiting times bare the risk of completed work to become obsolete. Furthermore, high inventory in requirements engineering can be due to that a high number of extra features have been defined.
- Inventory slows down the whole development process. Consider the example of a highway, if the highway is overloaded with cars then the traffic moves slowly.
- High inventory causes stress in the organization [10].

Lean manufacturing has drastically increased the efficiency of product development and the quality of products in manufacturing (see for example [4]). When implemented in software development lean led to similar effects (cf. [10, 11]). Even though lean principles are very promising for software development, the introduction of lean development is very hard to achieve as it requires a large shift in thinking about software processes. Therefore, an attempt to change the whole organization at once often leads to failure. This has been encountered when using lean in manufacturing [12] and software development [8].

To avoid the risk of failure when introducing lean our method helps the organization to arrive at a lean software process incrementally through continuous improvements. The method relies on the measurement of different inventories as well as the combined analysis of inventory measurements. The focus on inventory measurement is motivated by the problems caused by inventories discussed earlier. Furthermore, inventories also show the absence of lean practices and thus can be used as support when arguing for the introduction of the principles. In the analysis of the inventories a system thinking method is proposed as lean thinking requires a holistic view to find the real cause of problems. That is, not only single parts of the development process are considered, but the impact of problems (or improvement initiatives) on the overall process have to be taken into consideration.

Initial feedback on SPI-LEAM was given from two software process improvement representatives at Ericsson AB (see [13]). The objective was to solicit early feedback on the main assumptions and steps of SPI-LEAM from the company, which needs triggered the development of the method.

The remainder of the paper is structured as follows: Section 2 presents the related work on lean software development in general and measurement for lean software development in particular. Section 3 presents the Software Process Improvement through the Lean Measurement (SPI-LEAM) Framework. Section 4 presents a preliminary evaluation of the method. Section 5 discusses the proposed method with focus on comparison to related work, practical implications, and research implications. Section 6 concludes the paper.

2. Related Work

2.1. Lean in Software Engineering

Middleton [8] conducted two industrial case studies on lean implementation in software engineering, and the research method used was action research. The company allocated resources of developers working in two different teams, one with experienced developers (case A) and one with less experienced developers (case B). The responses from the participants was that initially the work is frustrating as errors become visible almost immediately and are returned in the beginning. In the long run though the number of errors dropped dramatically. After the use of the lean method the teams were not able to sustain the lean method due to organizational hierarchy, traditional promotion patterns, and the fear of forcing errors into the open.

Another case study by Middleton et al. [11] studied a company practicing lean in their daily work for two years. They found that the company had many steps in the process not being value-adding activities. A survey among people in the company showed that the majority supports lean ideas and thinks they can be applied to software engineering. Only a minority (10 %) is not convinced of the benefits of lean software development. Statistics collected at the company show a 25 % gain in productivity, schedule slippage was reduced to 4 weeks from previously months or years, and time for defect fixing was reduced by 65 % - 80 %. The customer response on the product released using lean development was overwhelmingly positive.

Perera and Fernando [14] compared an agile process with a hybrid process of agile and lean in an experiment involving ten student projects. One half of the projects was used as a control group applying agile processes. A detailed description of how the processes differ and which practices are actually used was not been provided. The outcome is that the hybrid approach produces more lines of code and thus is more productive. Regarding quality, early in development more defects are discovered with the hybrid process, but the opposite trend can be found in later phases, which confirms the findings in [8].

Parnell-Klabo [15] followed the introduction of lean and documented lessons learned from the introduction. The major obstacles in moving from agile are to obtain open office space to locate teams together, gain executive support, and training and informing people to reduce resistance of change. After successfully changing with the help of training workshops and use of pilot projects positive results have been obtained. The lead-time for delivery has been decreased by 40 % - 50 %. Besides having training workshops and pilots sitting together in open office-landscapes and having good measures to quantify the benefits of improvements are key.

2.2. Lean Manufacturing and Lean Product Development

Lean principles initially focused on the manufacturing and production process and the elimination of waste within these processes that does not contribute to the creation of customer value. Morgan and Liker [16] point out that today competitive advantage cannot be achieved by lean manufacturing alone. In fact most automotive companies have implemented the lean manufacturing principles and the gap in performance between them is closing. In consequence lean needs to be extended to lean product development, not only focusing on the manufacturing/production process. This trend is referred to as lean product development which requires the integration of design, manufacturing, finance, human resource management, and purchasing for an overall product [16]. Results of lean product development are more interesting for software engineering than the pure manufacturing part as the success of software development highly depends on an integrative view as well (requirements, design and architecture, motivated teams, etc.), and at the same time has a strong product focus.

Morgan and Liker [16] identified that inventory is influenced by the following causes: batching (large hand-overs of, for example, requirements), process and arrival variation, and unsynchronized concurrent tasks. The causes also have a negative effect on other wastes: batching leads to overproduction; process and arrival variation leads to overproduction and waiting; and unsynchronized tasks lead to waiting. Thus, quantifying inventory aids in detecting the absence of lean principles and can be mapped to root causes. As Morgan and Liker [16] point out their list of causes is not complete. Hence, it is important to identify the causes for waste after detecting it (e.g. in form of inventories piling up).

Karlsson and Ahlströhm [17] identified hinders and supporting factors when introducing lean production in a company in an industrial study. The major hinders are: (1) It is not easy to create a cross-functional focus as people feel loyal to their function; (2) Simultaneous engineering is challenging when coming from sequential work-processes; (3) There are difficulties in coordinating projects as people have problems understanding other work-disciplines; (4) It is challenging to manage the organization based on visions as people were used to detailed specifications and instructions; (5) The relationship to customers is challenging as cost estimations are expected, even in a highly flexible product development process. Factors helping the introduction of lean product development are: (1) Lean buffers in schedules; (2) Close cooperation with customers in product development to receive feedback; (3) For moving towards a new way of working successfully high competence engineers have to be involved; (4) Commitment and support from top management is required; (5) regular face-to-face meetings of managers from different disciplines.

Oppenheim [18] presents an extension of the value-stream mapping process to provide a comprehensive framework for the analysis of the development flow in lean product development. The framework is split into different steps. In the first step a takt-period is selected. A takt is a time-box in which different tasks are fulfilled and integrated. The second step is the creation of a current-state-map of the current process. The current-state map enables the identification of wastes and is the basis for the identification of improvements (e.g. the relations between waiting times and processing times become visible). Thereafter, the future-state-map is created which is an improved version of the map. Oppenheim stresses that all participants of the value-stream mapping process must agree to the future-state-map. After having agreed to the map the tasks of the map are parsed into the takt times defined in the first step. The last step is the assignment of teams (team architecture) to the different tasks in the value stream map. A number of success measures have been defined for the proposed approach: Amount of throughput time cut in comparison to competitors or similar completed programs; Amount of waste removed in the value-stream map (time-units or monetary value); Deviation of the planned value stream and the real value stream; Morale of the teams in form of a survey. Furthermore, the article points out that the goal of lean is to become better, cheaper, and faster. Though the reality often was that cheaper and faster was achieved on the expense of better (i.e. quality). One possible reason could be that no combined analysis of different measures was emphasized. Instead, measures were proposed as separate analysis tools (see Maskell and Baggaley [19] for analysis tools in the manufacturing context), but there is no holistic measurement approach combining individual measures to achieve a comprehensive analysis.

3. SPI-LEAM

The framework is based on the QIP [20] and consists of the steps shown in Figure 1. The steps according to Basili [20] are 1) Characterize the Current Project, 2) Set Quantifiable Goals and Measurements, 3) Choose Process Models and Methods, 4) Execute Processes and Collect and Validate Collected Data, 5) Analyze Collected Data and Recommend Improvements, and

6) Package and Store Experiences Made. The main contribution of this paper is to present a solution to step 2) for lean software development. The steps marked gray apply our method to achieve continuous improvement towards a lean software process.

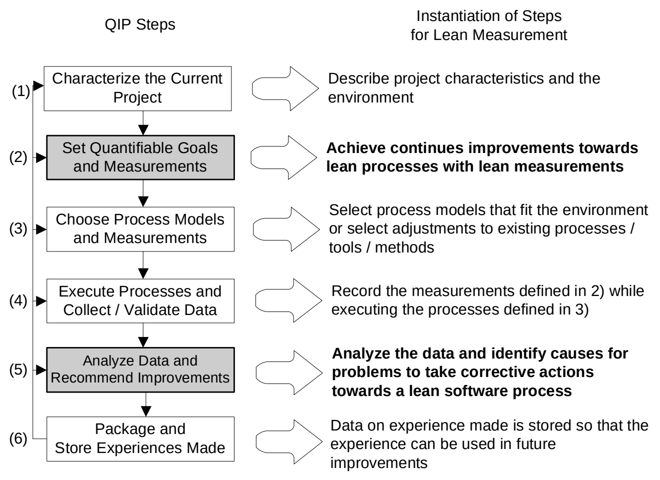

QIP Steps
Instantiation of Steps for Lean Measurement
(1) Characterize the Current Project
Describe project characteristics and the environment
(2) Set Quantifiable Goals and Measurements
Achieve continues improvements towards lean processes with lean measurements
(3) Choose Process Models and Measurements
Select process models that fit the environment or select adjustments to existing processes / tools / methods
(4) Execute Processes and Collect / Validate Data
Record the measurements defined in 2) while executing the processes defined in 3)
(5) Analyze Data and Recommend Improvements
Analyze the data and identify causes for problems to take corrective actions towards a lean software process
(6) Package and Store Experiences Made
Data on experience made is stored so that the experience can be used in future improvements

Figure 1: SPI-LEAM: Integration of QIP and Lean Principles

The steps are executed in an iterative manner so that one can always go back to previous steps and repeat them when needed. The non-expended steps are described on a high level and more details are provided for the steps expanded in relation to lean. The expansion of the second step (Set Quantifiable Goals and Measures) is explained in Section 3.2, and the expansion of the fifth step (Analyze Collected Data and Recommend Improvements) in Section 3.3.

1. *Characterize the Current Project:* In the first step, the project characteristics and the environment are characterized. This includes characteristics such as application domain, process experience, process expertise, and problem constraints [20].

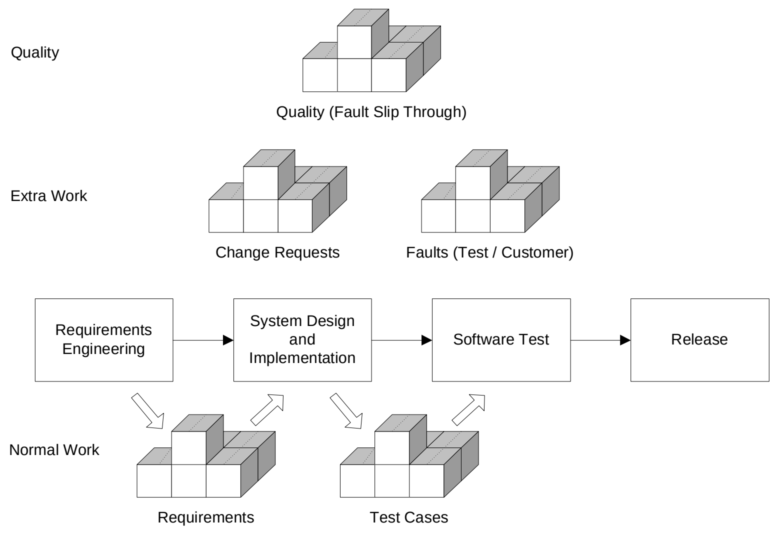


Figure 2: Inventories in the Software Process

2. *Set Quantifiable Goals and Measurements:* It is recommended to use the goal-question-metric approach [2] to arrive at the goals and associated measurement. The goal to be achieved with our method is to enable continuous software process improvement leading to a lean software process. The emphasis is on achieving continuous improvement towards lean software processes in order to avoid the problems when introducing an overall development paradigm and getting acceptance for the change, as seen in [8, 12]. The goals of achieving a lean process are set by the researchers to make the purpose of the method clear. When applying the method in industry it is important that the goals of the method are communicated to industry. In order to achieve the goals two key questions are to be answered, namely 1) what is the performance of the development process in terms of inventories, and 2) what is the cause of performance problems? The answer to the first question should

capture poor performance in terms of different inventories. Identifying high inventory levels will help to initiate improvements with the aim of reducing the inventories and by that avoiding the problems caused by them (see Section 1). The second question aims at identifying why the inventory levels are high. This is important to initiate the right improvement. To identify the inventories in the development process one should walk through the development process used in the company (e.g,. using value stream mapping aiming at identifying waste from the customers' perspective [7]). This is an important step as not all companies have the same inventories. For example, the company studied requires an inventory for product customizations. For a very generic process, as shown in Figure 2, we can find the following inventories: requirements, test cases, change requests, and faults. The inventories for faults and fault-slip-through represent the quality dimension. A detailed account and discussion of the inventories is provided in Section 3.2. The collection mechanism is highly dependent on the company in which the method is applied. For example, the collection of requirements inventory depends on the requirements tool used at the specific company.

3. *Choose Process Models and Methods:* This step decides which process model (e.g., waterfall approach, Extreme Programming (XP), SCRUM) to use based on the characteristics of projects and environment (Step 1). If, for example, the environment is characterized as highly dynamic an agile model should be chosen. We would like to point out that the process model is not always chosen from scratch, but that there

are established models in companies that need further improvement. Thus, the choice can also be a modification to the process models or methods and tools used to support the processes. For example, in order to establish a more lean process one approach could be to break down work in smaller chunks to get things faster through the development process.

4. *Execute Processes and Collect and Validate Collected Data:* When the process is executed measurements are recorded. In the case of SPI-LEAM the inventory levels have to be continuously monitored. Thereafter, the collected data has to be validated to make sure that it is complete and accurate.
5. *Analyze Collected Data and Recommend Improvements:* In order to improve performance in terms of inventory levels the causes for high inventories have to be identified. For understanding a specific situation and to evaluate improvement actions system thinking methods have been proven to be successful in different domains. Having found the cause for high inventory levels, a change to the process or the methods used has to be made.
6. *Package and Store Experiences Made:* In the last step the experience made has to be packaged, and the data collected needs to be stored so the experiences made are not lost and can be used for future improvements.

The following section provides an overview of how SPI-LEAM aids in continuously improving software processes to become more lean. This includes detailed descriptions of the expansions in steps two and five of the QIP.

3.1. Lean Measurement Method at a Glance

The first part of the method is concerned with setting quantifiable goals and measurements (Second step in the QIP). This includes the measurement of individual inventories. Thereafter, the measures of individual inventories are combined with each other, and with quality measurements (see Figure 3). How to measure individual inventories and combine their measurements with quality are explained in Section 3.2. For analysis purposes it is recommended not to use more than five inventories to make the analysis manageable. It is also important that at least one inventory focuses on the quality dimension (faults, fault-slip-through). Each measure should be classified in two different states, namely high inventory levels and low inventory levels. With that restriction in mind the company can be in $2^5 = 32$ different states (2 corresponding to high and low inventory levels, and 5 corresponding to the number of measurements taken). However, as the technique allows to measure on different abstraction levels, one individual inventory level (e.g. requirements) is derived from several sub-inventories (e.g. high level requirements, detailed requirements). Thus, we believe that companies should easily manage with a maximum of five inventories without neglecting essential inventories. Table 1 summarizes the goals, questions, and metrics according to the Goal-Question-Metric approach [2]. The inventories in the table are based on activities identified in the software engineering process (see e.g. [21]). As mentioned earlier each company should select the inventories relevant to their software processes.

The second part of the method is concerned with the analysis of the situation, i.e. the aim is to determine the causes for high inventory level

Table 1: Goal Question Metric for SPI-LEAM

Dimension	Specification
Goals	• Enable continuous software process improvement leading to a lean software process. • Avoid problems related to resistance of change by improving in a continuous manner.
Questions	• Q1: What is the performance of the development process in terms of inventories? • Q2: What is the cause of performance problems?

Metrics	• Requirements (Individual Inv.)
	  – High level Req. (Sub.-Inv.)
	  – Detailed Req.
	  – Req. in Design and Impl.
	  – Req. in Test
	• Test Cases (Individual Inv.)
	  – Unit Test (Sub.-Inv.)
	  – Function Test
	  – Integration Test
	  – System Test
	  – Acceptance Test
	• Change Requests (Individual Inv.)
	  – CR under Review
	  – Approved CRs
	  – CRs ready for Impact Analysis
	  – CRs in Test
	• Faults and Failures (Individual Inv.)
	  – Internal Faults and Failures (Test)
	  – External Faults and Failures (Customer)
	• Fault Slip Through (Quality)
	  – Req. Review Slippage
	  – Unit Test Slippage
	  – Function Test Slippage
	  – etc.

and quality problems. In other words, we want to know why we are in a certain state. Based on the analysis it is determined to which state one

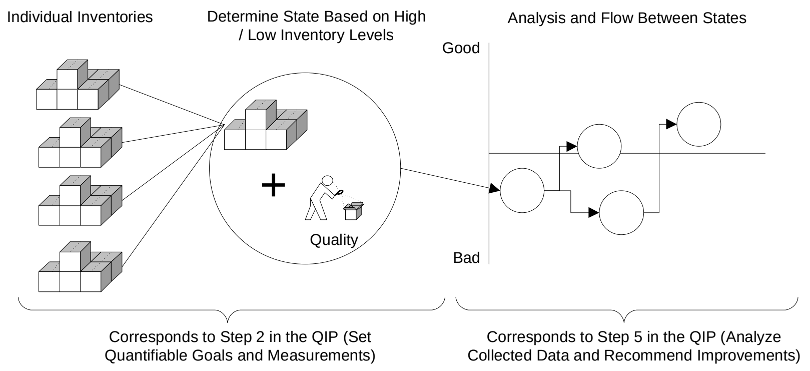

Individual Inventories
Determine State Based on High / Low Inventory Levels
Analysis and Flow Between States
Good
Bad
+
Quality
Corresponds to Step 2 in the QIP (Set Quantifiable Goals and Measurements)
Corresponds to Step 5 in the QIP (Analyze Collected Data and Recommend Improvements)

Figure 3: Method at a Glance

should move next. For example, which inventory level should be reduced first, considering the knowledge gained through the analysis. To make the right decisions the end-to-end process and the interaction of its parts are taken into consideration. As system theory in general and simulation as a sub-set of system theory methods allows to consider the interactions between individual parts of the process (as well as the process context) we elaborate on which methods are most suitable to use in Section 3.3.

3.2. Measure and Analyze Inventory Levels

The measurement and analysis takes place in the second step of the QIP as shown in the previous section. The measurement and analysis of inventory requires two actions:

- *Action 1:* Measure the individual inventories, i.e. change requests, faults, requirements, and test cases (see Figure 2). Each individual inventory (e.g., requirements) can have sub-inventory (e.g. high level requirements, detailed requirements, requirements in implementation, requirements in test) and thus these inventories are combined to a single inventory "requirements". The measurement of the inventories is done continuously to be able to understand the change of inventories over time. Inventories can also be split into work in process and buffer. The advantage is that this allows to determine waiting times, but it also requires to keep track of more data points.

- *Action 2:* Determine the overall state of the software process for a specific product combining the individual inventory levels. The combination of individual inventories is important as this avoids unintended optimization of measurements. That is, looking on one single inventory like requirements can lead to optimizations such as skipping requirements reviews in order to push the requirements into the next phase quicker. However, when combining inventory measures with each other and considering inventories representing the quality dimension will prevent this behavior. As a quality dimension we propose to use fault-slip through measurement [22] and the fault inventory.

3.2.1. Step 1: Measure Individual Inventories

Measurement Scale: In this step each of the inventories is measured individually considering the effort required for each item in the inventory. In manufacturing inventory is counted. In software development just counting the inventory is insufficient as artifacts in software development are too different. For example, requirements are of different types (functional requirement, quality requirement) and of different complexities and abstraction levels (see for example [13]). Furthermore, requirements might never leave the inventory as they are always valid (e.g. requirements on system response times).

Similar issues can be found for testing, like test cases that are automated for regression testing are different from manual test cases. As a consequence we propose to take into account the effort required to implement a requirement, or to run a test case. To make the importance of considering effort more explicit, take the example of a highway again. It is a difference if we put a number of trucks on a crowded highway, or a number of cars. In terms of requirements this means that a high effort requirement takes much more space in the development flow than small requirements. For each of the inventories there are methods available to estimate the effort:

- *Requirements:* Function points have been widely used to measure the size of systems in terms of requirements. This can be used as an input to determine the effort for requirements [23]. Another alternative is to estimate in intervals by classifying requirements as small, medium, or large. For each class of requirements an interval has to be set (e.g. in person days).

- *Test Cases:* Jones [21] proposed a model to estimate the effort of testing based on function point analysis. Another approach combines the number of test cases, test execution complexity, and knowledge of tester to estimate the effort of test execution [24].

- *Change Requests:* The effort of a change is highly influenced by the impact the change has on the system. Therefore, impact analysis can be used to provide an indication of how much effort the change requires [25].

- *Faults:* A method for corrective maintenance (e.g. fixing faults) has been evaluated by De Lucia et al. [26] considering the number of tasks needed for maintenance and the complexity of the system.

Describe Individual Inventory Levels: Individual inventories are broken down further into sub-inventories when needed. For example, requirements can be broken down into high level requirements, detailed requirements, requirements in design phase, and requirements in release. Similar, test cases can be broken down into unit test cases, integration test cases, system test cases, etc. The measurement of effort in inventory is done on the lowest level (i.e. high level requirements inventory, detailed requirements inventory, and so forth) and later combined to one single measure for requirements inventory. In Figure 4 the example for requirements (fictional data) and related sub-inventories is illustrated using a radar chart. We use this way of illustrating inventories throughout the whole analysis as it allows to see the distribution of effort between inventories in one single view very well. Based on this view it is to be determined whether the inventory level is high or low for the specific company. As mentioned earlier this depends on the capacity available to the company. Therefore, for each dimension one has to compare the capacity with the inventory levels. The situation in Figure 4 would suggest that there are clear overload situations for high level requirements and requirements in implementation. The inventory of requirements to be specified in detail is operating at its maximum capacity while there is free capacity for requirements in test.

Simulating Overload Situations: One should aim for a situation where the process is not overloaded, but the process should be stressed as much as possible. Figure 5 illustrates the situation of overload. If the load (inventory)

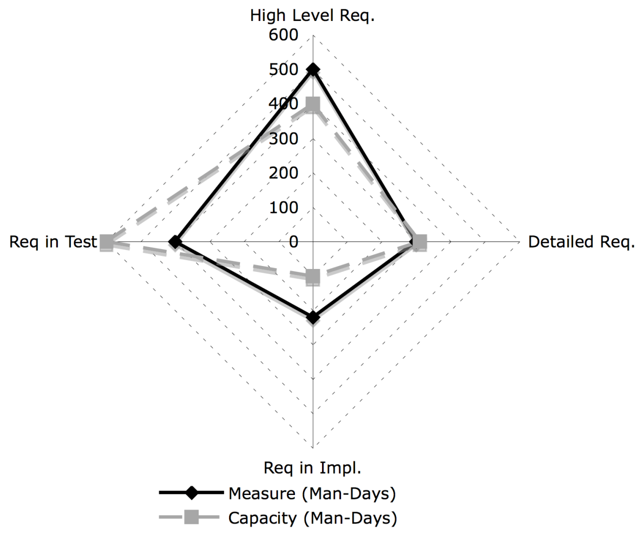


Figure 4: Measuring Requirements Inventory

is higher than the capacity then there is a high overload situation (in analogy to the highway there are more cars on the road than the road was designed for) and thus the flow in the process is jammed. A critical overload situation means close to complete standstill. If the capacity is almost completely utilized (the highway is quite full) then the flow moves slower, but still has a steady flow. Below that level (good capacity utilization) the resources are just used right, while when reducing the load too much an underload situation is created. To determine which are the thresholds for critical, high, and overload situations, queuing theory can be used. The inventory (e.g. requirements) represents the queue and the activity (e.g. system design and implementation) represents the server. An example of such a simulation for only requirements can be found in Höst et al. [27].

Set Inventory Level to High or Low: With the knowledge of when the organization is in an overload situation one can measure whether the inventory

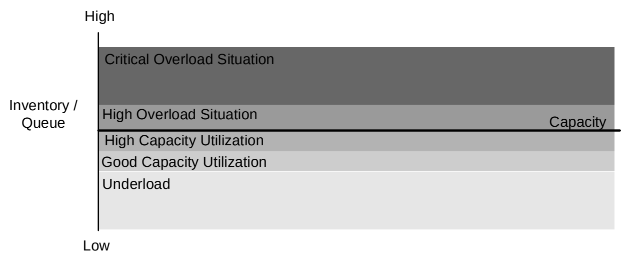

High
Inventory / Queue
Critical Overload Situation
High Overload Situation
Capacity
High Capacity Utilization
Good Capacity Utilization
Underload
Low

Figure 5: Good and Bad Inventory Levels

level is high or low. As we only allow two values in the further analysis (i.e. high and low) for reducing the complexity of the analysis we propose to classify inventories within the areas "good capacity utilization" and "underload" as low, and the ones above that as high. In order to combine the different sub-inventories for requirements into one inventory different approaches are possible, like:

- After having simulated the queues and knowing the zones the overall inventory level is high if the majority of sub-inventories is rated high.
- Sub-inventories are not independent (e.g. the queue for detailed requirements is influenced by the queue of high level requirements and the capacity of the server processing the high level requirements). Thus, more complex queuing networks might be used to determine the critical values for high and low.
- We calculate the individual inventory level II as the sum of the difference between capacity of the server for inventory i (C_i) and the actual measure for inventory i (A_i), divided by the number of sub-inventories n.

$$II = \frac{\sum_i^n (C_i - A_i)}{n} \tag{1}$$

If the value is negative then on average the company operates above their capacity. Thus, one should strive for a positive value which should not be too high as this would mean that the company operates with underload.

3.2.2. Determine the State of the Process Combining Inventories and Quality Measurement

As mentioned earlier it is good to restrict the number of states (e.g. to $2^5 = 32$) as this eases the analysis and it is possible to walk through and analyze the possible states in more depth. If this restriction is not feasible the number of states grows quite quickly, and with that the complexity of analysis. In order to illustrate the combined inventory level we propose to draw a spider web again with the average efforts for sub-inventories, the average capacity for each individual inventory and the rating as high or low. Even though we know the state for each inventory level the reason for drawing the diagram is that it allows to see critical deviations between inventories and capacity, as well as effort distributions between inventories. Besides the inventories it is important to consider the quality dimension. The reason is that this avoids an unintended optimization of measures. We propose to use the fault slip through (FST) and the number of faults to represent the quality dimension. The number of faults is a quality indicator for the product while the FST measures the efficiency of testing.

The FST measure helps to make sure that the right faults are found in the right phase. Therefore, a company sets up a test strategy determining which type of fault can (and should) be found in which test phase (e.g. function test). If the fault is found later (e.g. system test instead of function test) then the fault is considered a slip-through. Experience has shown that the measure in Equation 2 is one of the most useful [28]:

$$PIQ = \frac{SF}{PF} \tag{2}$$

The measure in Equation 2 shows the Phase Input Quality to a specific phase (phase X). SF measures the number of faults that are found in phase X, but should have been found earlier. PF measures the total number of faults found in phase X. When conducting this measurement it is also important to consider whether overall a high or low number of faults are found in phase X. In order to determine whether the testing strategy is in a good or bad state consider Figure 6. The figure shows four states based on 1) whether the overall number of faults found in phase X can be considered as high or low, and 2) whether the PIQ is high or low. Combining these dimensions leads to a high or low FST-figure. Whether the number of faults in phase X is high is context dependent and can, for example, be determined by comparing the number of faults found across different test phases.

The four dimensions in Figure 6 can be characterized as follows:

- *(No. Faults High, PIQ High)*: The fault-slip of faults to phase X is quite high, as well as the overall number of faults is high. This is an indicator for quality issues and low testing efficiency. We assign the value high to the FST measure.

- *(No. Faults High, PIQ Low)*: In this situation the test strategy is not

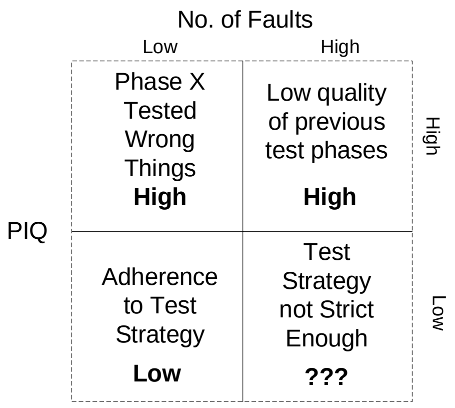

No. of Faults
Low
High
Phase X Tested Wrong Things
High
Low quality of previous test phases
High
High
PIQ
Adherence to Test Strategy
Low
Test Strategy not Strict Enough
???
Low

Figure 6: FST-Level

strict enough and should probably require that more faults should be found in earlier phases. As the data is based on a flawed test strategy the results should not be used as an indicator for process performance.

- *(No. Faults Low, PIQ High)*: Phase X probably tested the wrong things as one can assume that more faults are discovered considering a high fault-slip in earlier phases. We assign the value high to the FST measure.

- *(No. Faults Low, PIQ Low)*: The process adheres to the testing strategy and few faults are discovered which is an indicator of good quality. We assign the value low to the FST measure.

As effort is used throughout the method it is important to mention that FST can be transfered into effort as well. With the knowledge of average effort for fixing a fault found in a certain phase the improvement opportunity can be calculated.

In summary, the result of the phase are:

- A radar chart showing the average efforts and capacities related to each individual inventory, and the result of the improvement opportunity in terms of effort for faults found late in the process (FST).
- A description of the values of the inventory levels rated as either low or high with at least one inventory representing the quality dimension.

3.3. Analysis and Flow Between States

The analysis focuses on understanding the reasons for the current situation and finding the best improvement alternatives to arrive at an improved state. As a simple example consider a situation with one inventory (test cases) and the FST measure (see Figure 7). Analyzing the situation with just two measures shows that 1) no inventory measures should be taken without quality, and 2) combining measures allows a more sophisticated analysis than looking at different measures in isolation. Consider the situation in Figure 7 with one inventory (Test Cases) and the FST measure, both being labeled as either high or low based on the analysis presented before. Four different states are possible:

1. *(TC high, FST high)*: In this situation the company is probably in a critical situation as they put high effort in testing and at the same time testing is inefficient. Thus, this situation means that one has to explore the best actions for improvements. That is, one has to consider why the testing process and testing techniques lead to insufficient results. Possible sources might be found in the context of the process (e.g. requirements which form the basis for testing) or problems within the process (e.g. competence of testers).

2. *(TC high, FST low)*: This state implies that the company delivers good quality and puts much effort to do so in terms of test cases. Therefore, one does not want to move away from the low FST status, but wants to improve testing efficiency in order to arrive at the same result for FST without loosing quality. An option for improvement could be to switch to more automated testing so the effort per test case is reduced.
3. *(TC low, FST high)*: The test effort in terms of test cases is good, but there is low efficiency in testing. Either too little effort is spent on testing or the testing process and test methods need to be improved.
4. *(TC low, FST low)*: The state is good as testing is done in an efficient way with a low value for FST.

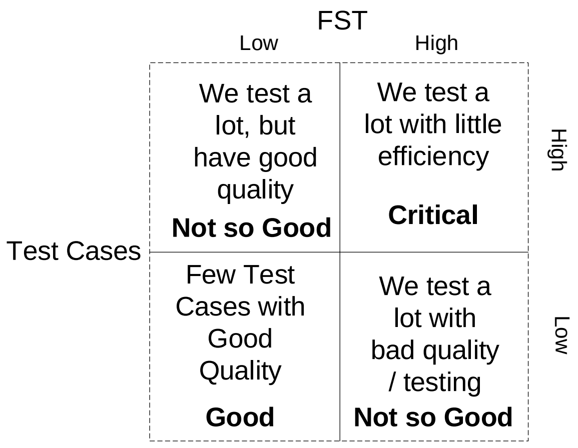

FST
Low
High
Test Cases
High
Low
We test a lot, but have good quality
Not so Good
We test a lot with little efficiency
Critical
Few Test Cases with Good Quality
Good
We test a lot with bad quality / testing
Not so Good

Figure 7: Analysis with Test Cases and FST

The analysis of the situation makes clear that it is important to include inventories representing the quality of the software product. Situations 1 and 2 are the same in terms of test case level, but lead to different implications when combined with the FST measure.

3.3.1. Analysis with n Inventories

With only two measures this seems to be obvious. Though, the outcome of the analysis will change when adding more inventories, and at the same time the analysis becomes more challenging. In order to characterize the situation of the company in terms of inventory levels the state of the company has to be determined. The state is defined as s tuple S of inventories s_i:

$$S := (s_1, s_2, s_3, ..., s_n),\ \ s_i \in \{high, low\} \tag{3}$$

What we are interested in is how improvement actions lead to a better state in terms of inventories. An ideal state would be one where all inventories have a good capacity utilization as defined in Figure 5. From an analysis point of view (i.e. when interpreting and predicting the behavior of development based on improvement actions) we assume that only one inventory changes at a time. When analyzing how to improve the state of inventories alternative improvement actions need to be evaluated. That is, the company should aim at reaching the desired state by finding the shortest path through the graph, the reason being to reduce the time of improvement impact. Figure 8 shows an example of state changes to achieve a desired state illustrated as a directed graph. The solid and dashed lines represent two different decisions regarding improvement actions. In the case of the graph decision one would be preferable to decision two as the desired state is achieved in fewer state changes, assuming the edges all have the same value.

Methods that can be used to support decision makers in the organization to make this analysis are presented in Section 3.3.3. In order to make the theoretical concepts in the case of n inventories more tangible the following

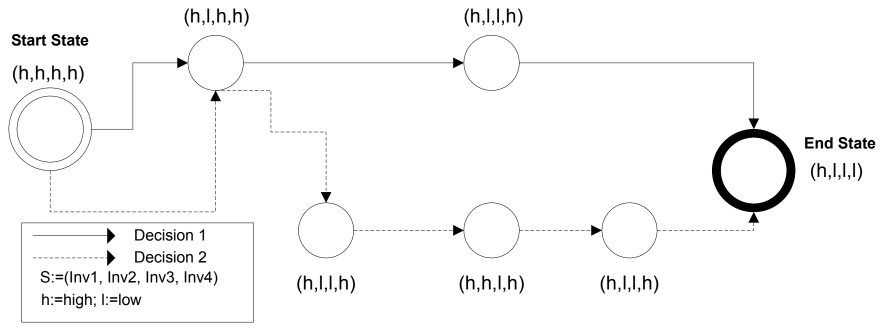


Figure 8: Improving the State of the Inventories

section presents an application scenario.

3.3.2. Application Scenario for Analysis with n States

The following inventories are considered in the application scenario and thus represent the state of the company S:

$$S := (Requirements,\ Change\ Requests,\ Faults,\ Test\ Cases,\ Fault\ Slip) \tag{4}$$

In the start state testing was done on one release and correct defects were found according to the test strategy. Furthermore, the release can be characterized as mature and thus it is stable, resulting in a low number of change requests. Though, the number of requirements is high as a project for the next release will be started. In addition to that testing has identified a number of faults in the product that need to be resolved. This leads to the following start state:

$$S_0 := (high,\ low,\ high,\ low,\ low) \tag{5}$$

The decision taken to achieve a more lean process in terms of inventory levels is to put additional resources on fixing the faults found in testing and wait with the new project until faults are resolved. Adding resources in testing leads to a reduction in the number of faults in the system.

$$S_1 := (high,\ low,\ low,\ low,\ low) \tag{6}$$

Due to fault fixing regression testing becomes necessary, which increases the number of test-cases to be run, leading to state S_2.

$$S_2 := (high,\ low,\ low,\ high,\ low) \tag{7}$$

Now that testing is handled the work starts on new requirements which leads to a reduction in requirements inventory. Furthermore, the value for FST could change depending on the quality of the tests conducted. This leads to a new current state:

$$S_3 := (low,\ low,\ low,\ high,\ low) \tag{8}$$

3.3.3. Analysis Support

Reasoning with different inventories to take the right actions is supported by the lean principle of "see the whole". In other words, a combined analysis of different parts of a whole (and the interaction between the parts) have already been considered when combining inventories and quality. As a solution for handling the analysis and evaluation of improvement actions different solutions are available which need to be evaluated and compared for their suitability in the lean context. Systems thinking as a method has been proven successful to conduct complex analyses. Three type of system approaches are common, namely hard systems, soft systems, and evolutionary systems.

- *Hard Systems:* These kind of systems are used for a quantitative analysis not considering soft factors [29]. The systems are usually described in quantitative models, such as simulation models. Candidates for analyzing the above problem are continuous simulations combined with queuing theory [30], or Discrete Event Simulations with queuing theory [27]. When repeating the activity in a continuous matter a major requirement on the model is to be simple and easily adjustable, but accurate enough. Fulfilling this requirement is the challenge for future research in this context.

- *Soft Systems:* Soft systems cannot be easily quantified and and contain interactions between qualitative aspects such as motivations, social interactions etc. Those problems are hard to capture in quantitative simulations and therefore some problems will only be discovered using soft system methodologies [29]. Therefore, they might be used as a complement to hard systems. In order to visualize and understand soft systems, one could make use of mind-maps or scenarios and discuss them during a workshop.

- *Evolutionary Systems:* This type of system applies to complex social systems that are able to evolve over time. However, those systems are very specific for social systems of individuals acting independently and are therefore not best suited from a process and workflow perspective.

After having decided on an improvement alternative the action is implemented and the improvements are stored and packaged. Thereby, it is important not just to describe the action, but take the lessons learned from the overall analysis as this will provide valuable data of behavior of the process in different improvement scenarios.

4. Evaluation

4.1. Static Validation and Implementation

The purpose of the static validation is to get early feedback from practitioners regarding improvements and hinders in implementing the proposed approach. Another reason for presenting the approach is to get a buy-in from the company to implement the approach [31]. In this case the method has been presented and discussed with two representatives from Ericsson AB in Sweden, responsible for software process improvement at the company. The representatives have been selected as they are responsible for identifying improvement potential in the company's processes, as well as to make improvement suggestions. Thus, they are the main stakeholders of such a method. The goal was to receive early feedback on the cornerstones of the methodology (e.g. the goals of the method; keeping work-load below capacity; combining different measurement dimensions; easy to understand representation of data), as well on limitations of the method. The following feedback was given:

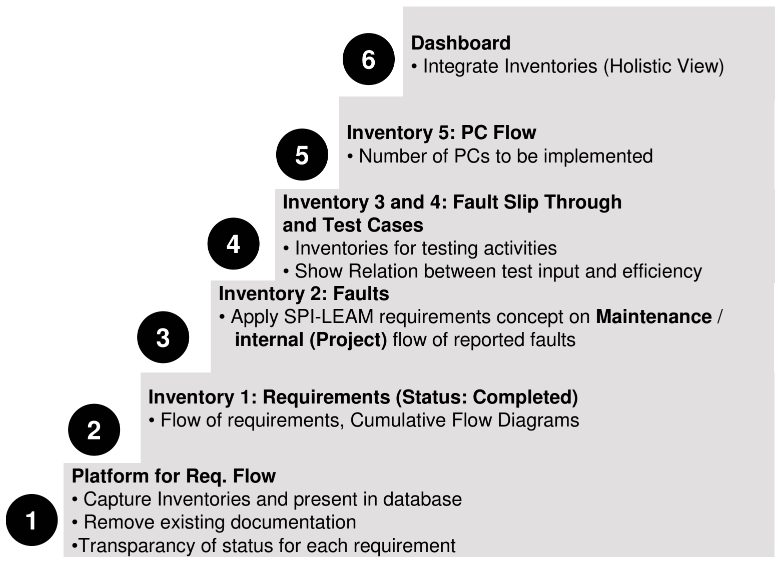
- The practitioners agree with the observation that the work-load should be below the capacity. Being below capacity is good as this, according to the practitioners experience, makes the development flow more steady. Furthermore, a lower capacity situation provides flexibility to fix problems in already released software products or in handling customization requests.

- When introducing inventory measures at the company the issue of optimization of measures was considered early on in the approach. For example, in order to reduce the level of inventory in requirements one could quickly write requirements and hand them over to the next phase to achieve measurement goals. In consequence, the company decided to consider inventories representing normal work (requirements flow) as well as quality related inventories (number of faults, and fault-slip-through). Furthermore, the company measures the number of requests from customers to provide individual customizations to their systems, which is an inventory representing extra work.

- The illustration of the data (capacity vs. load) in the form of radar charts was perceived as easy to understand by the practitioners, due to the advantages mentioned earlier. That is, one can gain an insight of several inventories at the same time, and observe how significant the deviation between capacity and load is.

The following limitation was observed by the practitioners: The method relies on knowing the capacity of the development in comparison to the work-load. The practitioners were worried that the capacity and work-load are hard to determine and are not comparable. For example, there is a high variance in developer productivity. Even though a team has N developers working X hours a day the real capacity is not equal to N*X hours. Furthermore, the work-load needs to be estimated as, for example, in terms of size of requirements. An estimation error would lead to an invalid analysis.

In summary, the practitioners perceived the method as useful in achieving a more lean software process. Based on the static validation and further discussions with the practitioners an implementation plan for the method was created (see Figure 9).



6
Dashboard
• Integrate Inventories (Holistic View)
5
Inventory 5: PC Flow
• Number of PCs to be implemented
4
Inventory 3 and 4: Fault Slip Through and Test Cases
• Inventories for testing activities
• Show Relation between test input and efficiency
3
Inventory 2: Faults
• Apply SPI-LEAM requirements concept on Maintenance / internal (Project) flow of reported faults
2
Inventory 1: Requirements (Status: Completed)
• Flow of requirements, Cumulative Flow Diagrams
1
Platform for Req. Flow
• Capture Inventories and present in database
• Remove existing documentation
•Transparancy of status for each requirement

Figure 9: Implementation Steps of SPI-LEAM

The first two steps are related to requirements inventories. The rational for starting with requirements was that implemented requirements provide value to the customer and thus are of highest priority in the implementation of SPI-LEAM.

The first step of the implementation plan is the creation of a web-platform to capture the requirements flow. The platform lists the requirements in each phase and allows the practitioners to move requirements between phases. When a requirement is moved from one phase to another the date of the movement is registered. Thus, the inventory level at any point in time for each phase of development can be determined.

The second step is the analysis of the requirements level by measuring the number of requirements in different phases. Complementarity to the inventory levels the data is also the basis to conduct analysis to get further insights into how requirements evolve during the development life-cycle. Cumulative flow diagrams were used to visualize the flow and throughput of development. In addition, the time requirements stayed in different phases were illustrated in the form of box-plots. The first and second steps have been completed and preliminary data collected in these step is presented in Section 4.2.

In the third step the number of faults is measured in a similar manner as the requirements flow. The company captures the flow of fixing the faults (e.g. number of faults registered, in analysis, implementation, testing, and regression testing). With this information one can calculate the number of requirements in different phases and construct cumulative flow diagrams as well. The flow of fixing faults is separated between internal (faults discovered by tests run by the company) and external (faults reported by the customer).

The fourth step in the staircase is concerned with measuring the number of test cases and the fault-slip through. An analysis for a combination of these inventories is shown in Figure 7.

The fifth step measures the number customization requests by customers. The development of customization follows an own flow which can be analyzed in-depth in a similar fashion as the flow of requirements (Step 2) and faults (Step 3).

In the last (sixth) step a dashboard is created which integrates the analysis of the individual inventory measurements on a high level (i.e. capacity vs. actual level of inventory) in form of a radar chart. To conduct an in-depth analysis a drill-down is possible. For example, if the inventory for requirements is high then one can investigate the requirements flow in more detail.

4.2. Preliminary Data

The requirements inventory was measured for a large-scale telecommunication product developed at Ericsson AB. The product was written in C++ and Java and consisted of over five million lines of code (LOC). Figure 10 shows an individual and moving range (I-MR) chart for the inventory of requirements in implementation (design, coding, and testing) over time. Due to confidentiality reasons no actual values can be reported. The continuous line in the figure shows the mean value while the dashed lines show the lower and upper control limits. The upper and lower control limits are $+/-$ two or three standard deviations away from the mean. A value that is outside the control limits indicates an instability of the process. The graph on the top of Figure 10 shows the individual values, while the graph on the bottom shows the moving range. The moving range is calculated as $MR = |X_i - X_{i-1}|$, i.e. it shows the size of the change in X between data point i and data point $i - 1$.

The figure for the individual values shows a large area of data points outside the upper control limit. In this time period the data indicates a very high inventory level. When presenting the inventory data to a development unit the participants of the unit confirmed that development was fully uti-

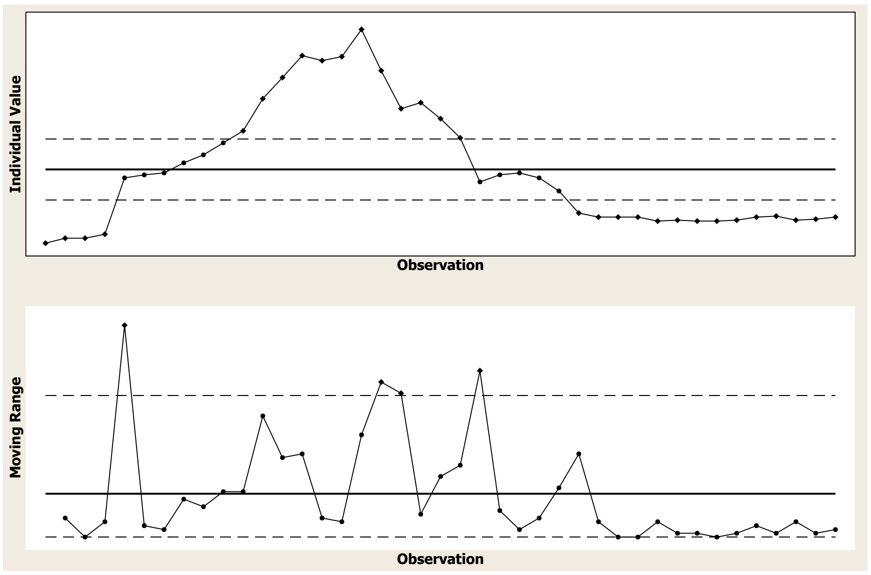


Figure 10: Inventory of Requirements in Implementation over Time (Observation = Time)

lized and people felt overloaded. This also meant that no other activity (e.g. refactoring) did take place besides the main product development. The opposite can be observed on the right-hand side of the figure where the data points are below the control limit. In this situation most of the requirements passed testing and were ready for release, which meant that the development teams idled from a main product development perspective. This time was used to solely concentrate on activities such as refactoring. To determine the capacity of development (and with that acceptable inventory levels) we believe it is a good and practical approach to visualize the inventories and discuss at which times the development teams felt overloaded or underloaded.

It is also interesting to look at the figure showing the moving range. A large difference between two data-points is an indication for batch-behavior, meaning that many requirements are handed over at once. Large hand-overs also constitute a risk of an overload-situation. From a lean perspective one should aim at having a continuous and steady flow of requirements into the development, and at the same time delivering tested requirements continuously.

Collecting inventory data in a continuous manner also enables the use of other lean analysis tools, such as cumulative flow diagrams [32, 33]. They allow to analyze the requirements flow in more detail with respect to throughput and cycle times. The graph in Figure 11 shows a cumulative flow diagram which is based on the same data as the control charts. The top line represents the total number of requirements in development. The line below that shows the number of requirements that have been detailed and handed over to implementation. The next line shows the number of requirements handed over to node test, and so forth. The difference between two lines at a certain point in time shows the inventory. The lines themselves represent hand-overs.

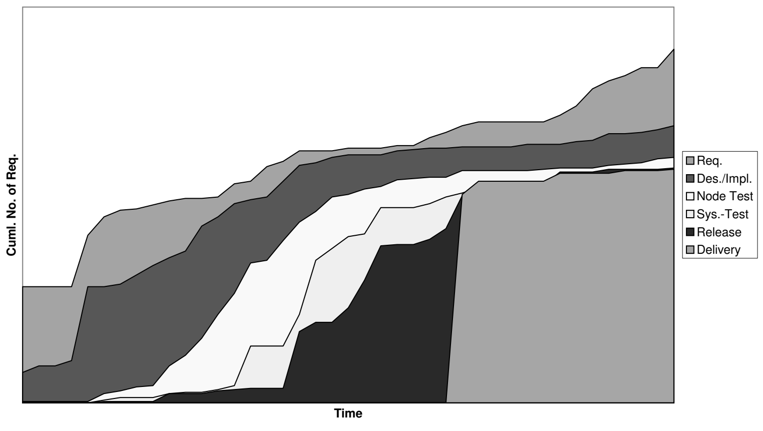


Figure 11: Cumulative Flow Diagram

Cumulative flow diagrams provides further information about the development flow that can be used complementary to the analysis of inventory levels shown in Figure 10. The cumulative flow diagram shows velocity/throughput (i.e. the number of requirements handed over per time unit). Additionally it provides information about lead-times (e.g. the majority of the requirements is handed over to the release after 2/3 of the overall development time has passed). The batch behavior from the moving range graph can also be found in the cumulative flow diagram (e.g. large handover from node-test to system test in the middle of the time-line). A detailed account of measures related to development flow and the use of cumulative flow diagrams in the software engineering context can be found in [34].

4.3. Improvements Towards Lean

Based on the observations from measuring the requirements flow the company suggested a number of improvements to arrive at a more lean process. Two examples are provided in the following:

- *From Push to Pull:* A reason for the overload situation is that the requirements are pushed into development by allocating time-slots far ahead. Due to the analyses the desire is to change to a pull-approach. This should be realized by creating a buffer of prioritized requirements ready for implementation from which the development teams can pull. Thereby, the teams get more control of the work-load which is expected to help them in delivering in a more continuous manner.

- *Intermediate Release Versions:* Before having the final delivery of the software system with agreed scope intermediate releases should be established that that have the quality to be potentially released to the customer. Thereby the company wants to achieve to 1) have high quality early in development, and 2) have a motivator to test and integrate earlier.

It is important to emphasize that only a part of SPI-LEAM has been implemented so far. A more holistic analysis will be possible when having the other inventory measurements available as well. One interesting analysis would be to plot all inventories in graphs such as the ones presented in Figures 10 and 11 and have a look at them together. For example, an overload in the requirements flow would explain a slow-down in the maintenance activities.

5. Discussion

5.1. Comparison with Related Work

A comparison with the related work allows to make two observations which indicate the usefulness of SPI-LEAM as an extension over existing measurements in the lean context.

First, Morgan and Liker [16] identified a mapping between root causes and waste. An interesting implication is that the occurrence of waste in fact points to the absence of lean practices. In consequence SPI-LEAM is a good starting point to identify wastes in development continuously. Based on the results provided by SPI-LEAM an undesired behavior of the development becomes visible and can be further investigated with techniques such as root cause analysis. This can be used as a motivation for management to focus on the implementation of lean practices. The transparency of the actual situation can also provide insights helping to overcome hindering factors, such as the ones reported in [17].

Secondly, Oppenheim [18] pointed out that it is important to not only focus on speeding up the development to achieve better cycle times for the expense of quality. To reduce this risk we proposed to provide inventory measurements that can be used to evaluate cycle times and throughput (see cumulative flow diagrams) and inventories related to quality together (flow of fixing faults). If, for example, cycle time is shortened by reducing testing this will show positively in the cycle time, but negatively in the fault-slip-through and the number of faults reported by the customers.

5.2. Practical Implications

The flexibility of the method makes it usable in different contexts. That is, the method allows companies to choose inventories that are relevant for their specific needs. Though, the only restriction is that at least one inventories should be considered that represents the quality dimension. The risk of optimizing measures on the cost of quality is thereby reduced. In addition, we presented different alternatives of analysis approaches (hard systems, soft systems, evolutionary systems) that companies can use to predict the effect of improvement actions.

Criticism was raised regarding the determination of capacity/work-load by the practitioners. In order to determine the right work-load, simulation can be used, in particular queuing networks and discrete event simulation (a theoretical description can be found in [35, 36]). An example of the application of discrete event simulation with queuing networks to determine bottlenecks in the software engineering context is shown in [27]. As mentioned earlier, there are also alternative approaches that can be used which are easier and faster to realize (see description of soft systems in Section

3.3.3). Another approach is to plot inventory levels based on historical data and have a dialog with development teams and managers about the workload situation over time.

The practitioners also perceived the analysis and prediction of the effect of improvement actions (as shown in Figure 8) as theoretical, making them feel that the method might not be practical. However, this was due to the way of illustrating the movement between states as a directed graph. In practice, one would provide a view of the inventory levels over time illustrating them as a control chart/cumulative flow diagram as shown in Section 4.2. Doing so allowed the practitioners to identify a departure from lean practices. For example, the cumulative flow diagram derived from the inventory data showed that the system was not built in small and continuous increments. This observation helped to raise the need for a more continuous flow of development and led the company to take actions accordingly (see Section 4.3). In other words, SPI-LEAM provides facts that allow a stronger position when arguing why specific practices should receive more attention.

Considering the feedback from industry on the method it seems a promising new approach to continuously improve software processes to become more lean. As the method is an instantiation of the QIP one can leverage of the benefits connected to that paradigm (e.g. having a learning organization due to keeping track of the experiences made, see last step in Figure 1).

5.3. Research Implications

The related work shows that lean software engineering has been evaluated as a whole, i.e. it is not clear which tools have been applied. Furthermore, the benefit of single tools from the lean tool-box have not been evaluated so far to learn about their benefits in software engineering contexts. Such applications also show how the methods have to be tailored (see, for example, capacity discussion in Section 5.2). In general there are only few studies in software engineering and more studies are needed which describe the context in detail in which lean was applied, as well as how lean was implemented in the companies.

Concerning SPI-LEAM, there is a need to evaluate what kind of improvement decisions are proposed and implemented based on the measurement results derived by the method. This has to be followed up in the long run to see whether continuous improvements will be achieved in terms of making the software process more lean. In addition, SPI-LEAM has to be applied in different contexts to learn what kind of inventories are the most relevant in specific contexts. For example, how does SPI-LEAM differ between different agile processes (SCRUM or eXtreme Programming) and different complexities (large products with many development teams vs. small products with few development teams).

In conclusion, we see a high potential of lean practices improving software engineering. However, there is very little work done in this area so far.

6. Conclusion

This paper presents a novel method called Software Process Improvement through Lean Measurement (SPI-LEAM). The goals of the method are to 1) enable continuous software process improvement leading to a lean software process; and 2) avoid problems related to resistance of change by improving in a continuous manner. The method combines the quality improvement paradigm with lean measurements.

The method is based on measuring inventories representing different dimensions of software development (normal development work, extra work, and software quality). It was exemplified how the method is used. Feedback from industry and the discussion of the method leads to the following conclusions:

- SPI-LEAM is flexible as it allows companies to choose inventories and analysis methods fitting their needs in the best way. Thus, the method should be applicable in many different contexts.

- The practitioners who reflected on the method agreed on how we approached the problem. They, for example, observed that the inventories should be below the capacity level. Furthermore, they agreed on the need to conduct a combined analysis of inventories to have a complete view of the current situation. That is, the risk of optimizing measures is drastically reduced.

In future work the impact of the method on software process improvement activities is needed. The method focused on the overall process life-cycle. However, the ideas could be useful for a team working on a specific development task, such as the visualization of throughput for a single team, and a measure of cycle-time. This allows each team to determine its own capability level. Furthermore, the analysis of related work showed that generally more research on lean tools is needed.

References

[1] CMMI-Product-Team, Capability maturity model integration for development, version 1.2, Tech. rep., CMU/SEI-2006-TR-008 (2006).

[2] V. R. Basili, Quantitative evaluation of software methodology, Tech. rep., University of Maryland TR-1519 (1985).

[3] V. R. Basili, S. Green, Software process evolution at the sel, IEEE Software 11 (4) (1994) 58–66.

[4] D. Cumbo, E. Kline, M. S. Bumgardner, Benchmarking performance measurement and lean manufacturing in the rough mill, Forest Products Journal 56 (6) (2006) 25 – 30.

[5] J. P. Womack, D. T. Jones, Lean thinking : banish waste and create wealth in your corporation, Free Press Business, London, 2003.

[6] S. Shingo, Study of ”Toyota” production system from the industrial engineering viewpoint, Japanese Management Association, 1981.

[7] M. Poppendieck, T. Poppendieck, Lean Software Development: An Agile Toolkit (The Agile Software Development Series), Addison-Wesley Professional, 2003.

[8] P. Middleton, Lean software development: Two case studies, Software Quality Journal 9 (4) (2001) 241–252.

[9] K. Petersen, C. Wohlin, D. Baca, The waterfall model in large-scale development - state of the art vs. industrial case study, in: Proceed-

ings of the 10th International Conference on Product Focused Software Development and Process Improvement, 2009, p. in submission.

[10] T. Morgan, Lean manufacturing techniques applied to software development, Master's thesis, Master Thesis at Massachusetts Institute of Technology (April 1998).

[11] P. Middleton, A. Flaxel, A. Cookson, Lean software management case study: Timberline inc, in: Proceedings of the 6th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2005), 2005, pp. 1–9.

[12] R. T. Pascale, Managing on the edge : how the smartest companies use conflict to stay ahead, Simon and Schuster, New York, 1990.

[13] T. Gorschek, C. Wohlin, Requirements abstraction model, Requir. Eng. 11 (1) (2006) 79–101.

[14] G. I. U. S. Perera, M. Fernando, Enhanced agile software development hybrid paradigm with lean practice, in: Proceedings of the International Conference on Industrial and Information Systems (ICIIS 2007), 2007, pp. 239–244.

[15] E. Parnell-Klabo, Introducing lean principles with agile practices at a fortune 500 company, in: Proceedings of the AGILE Conference (AGILE 2006), 2006, pp. 232–242.

[16] J. M. Morgan, J. K. Liker, The Toyota product development system: integrating people, process, and technology, Productivity Press, New York, 2006.

[17] C. Karlsson, P. Ahlströhm, The difficult path to lean product development, Journal of Product Innovation Management 13 (4) (2009) 283–295.

[18] B. W. Oppenheim, Lean product development flow, Systems Engineering 7 (4) (2004) 352–376.

[19] B. Maskell, B. Baggaley, Practical lean accounting: a proven system for measuring and managing the lean enterprise, Productivity Press, 2004.

[20] V. R. Basili, The experience factory and its relationship to other quality approaches, Advances in Computers 41 (1995) 65–82.

[21] C. Jones, Applied software measurement: assuring productivity and quality, McGraw-Hill, New York, 1997.

[22] L.-O. Damm, L. Lundberg, C. Wohlin, Faults-slip-through - a concept for measuring the efficiency of the test process, Software Process: Improvement and Practice 11 (1) (2006) 47–59.

[23] Çigdem Gencel, O. Demirörs, Functional size measurement revisited, ACM Transactions on Software Engineering and Methodology 17 (3).

[24] X. Zhu, B. Zhou, L. Hou, J. Chen, L. Chen, An experience-based approach for test execution effort estimation, in: Proceedings of the 9th International Conference for Young Computer Scientists (ICYCS 2008), 2008, pp. 1193 – 1198.

[25] M. Lindvall, K. Sandahl, Traceability aspects of impact analysis in

object-oriented systems, Journal of Software Maintenance 10 (1) (1998) 37–57.

[26] A. D. Lucia, E. Pompella, S. Stefanucci, Assessing effort estimation models for corrective maintenance through empirical studies, Information & Software Technology 47 (1) (2005) 3–15.

[27] M. Höst, B. Regnell, J. N. och Dag, J. Nedstam, C. Nyberg, Exploring bottlenecks in market-driven requirements management processes with discrete event simulation, Journal of Systems and Software 59 (3) (2001) 323–332.

[28] L.-O. Damm, L. Lundberg, C. Wohlin, A model for software rework reduction through a combination of anomaly metrics, Journal of Systems and Software 81 (11) (2008) 1968–1982.

[29] J. P. V. Gigch, System design modeling and metamodeling, Plenum Press, New York, 1991.

[30] P. Donzelli, G. Iazeolla, Hybrid simulation modelling of the software process, Journal of Systems and Software 59 (3) (2001) 227–235.

[31] T. Gorschek, P. Garre, S. Larsson, C. Wohlin, A model for technology transfer in practice, IEEE Software 23 (6) (2006) 88–95.

[32] D. G. Reinertsen, Managing the design factory: a product developers toolkit, Free, New York, 1997.

[33] D. Anderson, Agile management for software engineering: applying the theory of constraints for business results, Prentice Hall, 2003.

[34] K. Petersen, C. Wohlin, Measuring the flow of lean software development, in submission, http://kaipetersen79.googlepages.com/sSPE.pdf.

[35] G. Bolch, S. Greiner, H. D. Meer, K. S. Trivedi, Queueing networks and Markov chains: modeling and performance evaluation with computer science applications, 2nd Edition, Wiley, Hoboken, N.J., 2006.

[36] C. G. Cassandras, S. Lafortune, Introduction to discrete event systems, Kluwer Academic, Boston, 1999.

Kai Petersen is an industrial PhD student at Ericsson AB and Blekinge Institute of Technology. He received his Master of Science in Software Engineering (M.Sc.) from Blekinge Institute of Technology. Thereafter, he worked as a research assistant at University of Duisburg Essen, focusing on software product-line engineering and service-oriented architecture. His current research interests are empirical software engineering, software process improvement, lean and agile development, and software measurement.

Claes Wohlin is a professor of software engineering and the Pro Vice Chancellor of Blekinge Institute of Technology, Sweden. He has previously held professor chairs at the universities in Lund and Linköping. His research interests include empirical methods in software engineering, software metrics, software quality, and requirements engineering. Wohlin received a PhD in communication systems from Lund University. He is Editor-in-Chief of Information and Software Technology and member of three other journal editorial boards. Claes Wohlin was the recipient of Telenors Nordic Research Prize in 2004 for his achievements in software engineering and improvement of reliability for telecommunication systems.