PROCESS ANALYSIS OF A WATERFALL PROJECT USING REPOSITORY DATA

Michael VanHilst,* Shihong Huang,* and Hugh Lindsay**

Abstract

The study of repository data has yielded insight into the behaviour of developers and qualities of software artifacts. In this article, we use repository data to investigate the nature and quality of the development process itself. Specifically, we use repository data to identify sources of potential inefficiency and waste in process design and execution. Our analysis approach is motivated by principles of Lean product development and fills a gap in the literature for empirical process metrics. To illustrate the approach, we analyse data from a large industry project involving CMM level 3 and 4 teams in a waterfall process. Both basic and focused analyses are presented, with eight different views of progress, efficiency, and cost.

Key Words

Lean analysis, repository data, process metrics, and process improvement

1. Introduction

In this paper we present a new approach to process improvement based on empirical data and sound analysis practices. Traditional approaches have focused on process adoption and the behaviour of individuals. Our approach focuses on finding waste in the process itself. Through a variety of models presented in examples, we show how to find indicators of various types of waste, trace both their causes and effects, and provide baseline measurements against which to make comparisons when improvements to the process are applied.

The work in this article fills a gap in both academic and applied literature by demonstrating a range of metrics that specifically address process execution and design. We also take the novel approach of using only pre-existing data from two repository tools, namely task tracking and configuration management, to perform the analysis. In the analysis we show the value of time series graphs of effort and progress, and Little's Law graphs for Work-In-Process and Time-In-Process. While these graphs have occasionally been mentioned in other literature, in this article we show how to produce the graphs from empirical data and apply them to process improvement.

Our analysis is grounded in Lean thinking and the TQM/Six Sigma traditions of institutionalized learning and continuous improvement [1]. It does not assume a particular development process. The objective, rather, is to improve upon whatever process is in place. In Lean thinking, the standard process is what you are doing. Through observation and analysis, you continually seek to improve it. Change is pulled, not pushed. In a Lean improvement (Kaizan) you have specific reasons for each change and know what to expect [2]. Continuous measurement and analysis is vital to the process. But costly measurement is not sustainable. In this paper, we present an inexpensive and non-invasive approach to measurement.

To illustrate our approach, we present a real analysis using data from a large industry project. The project involved globally distributed teams of more than 100 developers in total. The teams coded in C, C++, and Java, were certified at CMM levels 3 and 4, and followed a waterfall process. The product being developed was a new release of an existing million+ line product. To protect the confidentiality of the source, more details are not available. But our message is not about the specific numbers. It is about practices for using whatever numbers you have to find improvements in an existing process.

The data for the analysis was extracted directly from the configuration management and task tracking systems. The data is empirical and can be collected with no cost added to the development team. For the analysis, we use both traditional management models and models from Lean analysis. We investigate issues raised in the initial analysis in greater depth using targeted models and statistical analyses, and apply simulation models to test hypotheses. While some of the analyses resemble analyses used in manufacturing, their use here is specific to the process of development.

In Section 2 we discuss the basis, motivation, and goals of the analysis while Section 3 describes the collection and preparation of the data. Section 4 describes the initial

* College of Engineering and Computer Science, Florida Atlantic University; e-mail: {mike, shihong}@cse.fau.edu

** General Dynamics C4 Systems; e-mail: Hugh.Lindsay@gdc4s.com

Recommended by Dr. L. Etzkorn
(10.2316/Journal.202.2011.1.202-2986)

analysis leading to a number of observations. Section 5 presents more in-depth analysis of an issue raised in the initial analysis of Section 4. Section 6 describes related work and its relationship to the material presented here. Section 7 presents our conclusion.

2. Motivation and Goals

Lean process improvement focuses on the reduction of waste. In Lean, waste is defined as anything that reduces value or adds cost. In applying Lean to development, the key intuition is that development is a knowledge activity, where cost and value apply to the production and consumption of knowledge. Lean product development (LPD) originates in Toyota's own practices of product design. It has been studied and documented by both the University of Michigan's Japan Technology Management Program [2, 3] and MIT's Lean Aerospace Initiative [4]. The Poppendiecks have written about applying LPD to software development [5, 6]. But they say little about metrics. This article is about using metrics to identify the source and measure the impact of waste. We begin by defining the seven forms of waste.

Taiichi Ohno and Shigeo Shingo, the pioneers of Toyota's Lean production system, identified seven forms of waste in manufacturing: defects, inventory, waiting, transportation, motion, over-production, and over-processing [7]. The seven wastes have direct correlates in software development.

Defects refer to defects that are not corrected at the time they occur. Defects require later rework – adding cost but no new value, or remain in the product – reducing value to the customer.

Inventory is work that is produced before it is needed, and sits for a period as inventory. In-process inventory adds throughput time with no increase in value. Items in inventory lose value and may increase cost at their time of use. Unused knowledge fades from memory and becomes increasingly expensive to recover. Queues of change requests and bug reports are a form of inventory.

Waiting occurs when work that could be done is not being done, due to delays or bottlenecks in other parts of the process. Testing is often a source of bottlenecks in software development. Defects that should be acted on must wait for testing to reveal them. Waiting for developers to be freed from other tasks or projects is another bottleneck.

Transportation refers to a variety of problems that occur when materials move from one group to another. Transportation in software development is called a hand-off. The recipient of a hand-off spends additional effort to acquire knowledge that the prior handlers of the artifact already had.

Motion refers to a workspace that is not well-organized with items ready-at-hand when needed. In software development, motion means searching for information, a problem that often accompanies task switching. Developers must recover information that they themselves may once have had, but have now forgotten, or at least have lost the train of thought needed to take the next step.

Over-production refers to things that are produced, but never used or needed. In software this can apply to features, options, and artifacts. The likelihood of over-production, or over-engineering, increases with the remoteness of the intended customer, or future. In the terminology of Agile development, over-production is called YAGNI, short for you ain't gonna need it.

Over-processing refers to activities that have little benefit in terms of customer value. In development, over-processing describes anything that would involve, cause, or require less work if addressed in some other way. Candidates could include ineffective testing, brittle architecture, outdated practices, or poor tool use. Developer education can remedy some forms of over-processing.

3. Repository Data

In a mature organization, configuration management is an integral part of code development. Code artifacts and the history of their changes over time are maintained in the configuration management system. These changes represent the work that has been done. In disciplined practice, all work must be associated with a work order that represents either a required change or a defect. Work items are tracked in bug and task tracking systems. Logs and records from these tools can be used to find activity begin and end times, overlap and time lags between activities, and various counts and frequencies.

With the data, we know for every task when it was first begun, when, on which files, and for how long work was done, and when it was actually completed. In the configuration management data, work on a task begins with the first development event and ends with the last development event associated with the task. Bug and task tracker Records give the type of task and the date on which the task's request or defect report was submitted. We also know on which days each developer had code checked out, and for which tasks. Assembling this data gives a detailed and comprehensive picture of the process.

Configuration management event log records give the artifact name, branch, developer, type of action, and date. Records from the bug and task tracker include the project, task, date submitted, and task type (requirement or defect). Correlation between the two data sources requires a task ID in both. Unfortunately, task IDs are not supported by most configuration management systems. The developers in our data use stylized ways of recording the task ID in branch names and in comments.

To perform the analyses, we created a derived table where each record represents a single artifact's involvement in a single task. The new record contains item, task ID, task type, date submitted, date of first work, date of last work, and the date merged back into system. Manually reported completion dates in the task tracker often conflicted with actual dates of last work and merge, and were ignored.

We also wanted to identify the date on which each defect was caused and in which task. To assign a date, we used the following assumption: the defect was caused in one of the artifacts that was later edited to correct the defect. Using this assumption, for all artifacts involved in

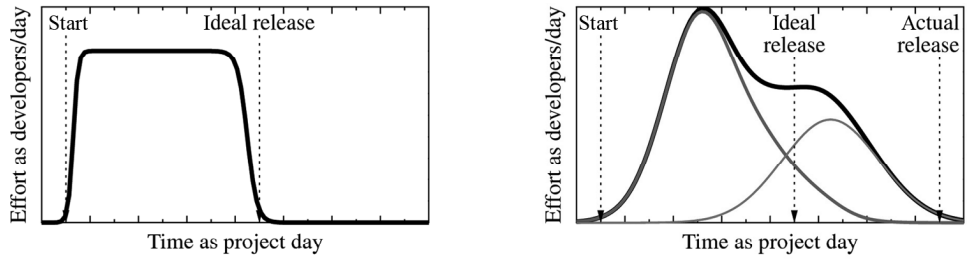

Figure 1. Development effort versus time for idealistic (left) and realistic (right) projects.

the repair, we looked at all changes completed and merged prior to the defect's submission. Taking a conservative approach, we chose the latest date from the set of candidate changes – the date nearest to the defect report.

For effort, we computed developer days. A developer is assumed to be active on a task on every day between the first and last change events associating that developer with that task. We then count each developer as active on any day on which they are active on at least one task. They are counted as actively developing if they are active on a change task, and actively repairing if they are active on a repair task. A developer could be active on two or more tasks at the same time, and even on one task of each kind; our data takes this into account. On the projects for which we have collected data, both situations do occur, but not often.

4. Basic Analysis

In the analysis presented here, the project followed a waterfall methodology and spanned 49 weeks from requirement freeze to first release. The project involved more than 100 globally distributed developers, though no more than 90 at any one time. There were several thousand artifacts and somewhat more than 1,000 tasks. The goal of this paper is to present a new approach to process improvement, using metrics. The project was chosen to illustrate the use of metrics. Details are specific to the teams involved and not necessarily representative of other company or industry projects. We withhold some details to protect the owner of the data. But many significant details are presented. Similar data from other projects appear in Kan *et al.* [8] and Mockus *et al.* [9].

4.1 Effort

In an ideal development project, effort should ramp up quickly, maintain a steady pace, and then ramp down at completion. A graph of that form, shown on the left in Fig. 1, represents optimal use of resources with minimal inefficiencies. The ideal graph resembles a boxcar function: straight up, straight across, and straight down. The ideal represents a continuous "flow" in Lean terminology [10] and is explained in a discussion of queue size and cycle time in a book by Reinertsen [11].

A more realistic graph of effort in real world development projects is shown on the right in Fig. 1. This line

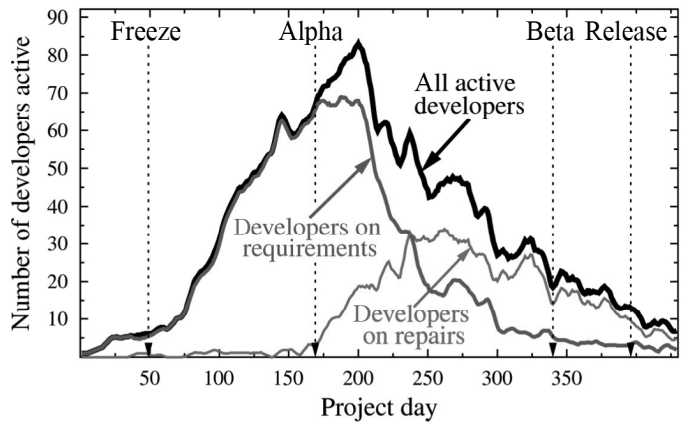

Figure 2. Project effort over time for requirements and for defects.

is the sum of two gradual humps. The first hump, for development, ramps up slowly as developers break free from other projects, possibly where they have been tied down fixing defects. Just as the first hump starts ramping down, a second hump ramps up. The second hump represents the rework effort to fix defects produced in the first hump. Not until the second hump winds down is the project ready for release. A question for analysis is: how can our project become more like the efficient boxcar ideal.

Figure 2 shows the same graph as Fig. 1 using the data from our project. The tallest curve counts developers active on any task. The middle-sized hump to the left was generated for requirement tasks only, while the smaller hump to the right considers only defect repair tasks. Notice that significant effort does not start until several weeks after requirements freeze and then ramps up only gradually. Significant rework effort does not start until alpha, 4 months into the project. Effort ramps down even more gradually. Later in the analysis, we will examine the issue of developer allocation in more depth.

The area under the first hump represents value-add, while the area under the second hump, rework, is waste. We compute the areas under the curves by summing the daily counts, which gives us total effort in developer days. In this project the area under the defect curve is very close to half the area under the requirement curve – defect repair accounts for 33% of total effort. Using this number as a baseline, we can compare this result with other projects and with efforts to improve.

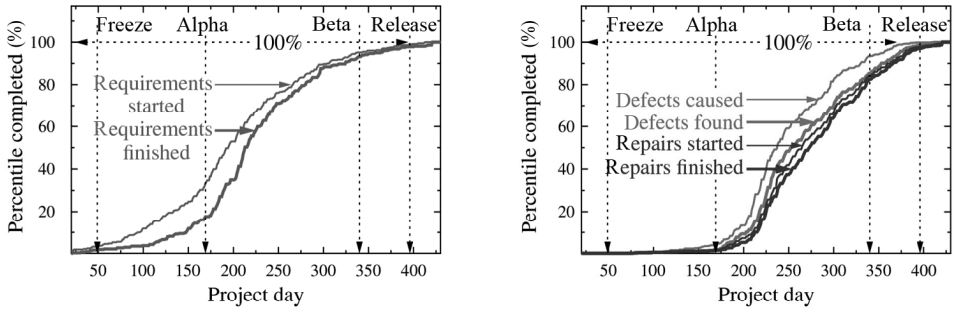

Figure 3. Cumulative progress curves for requirements (left) and repair (right) tasks.

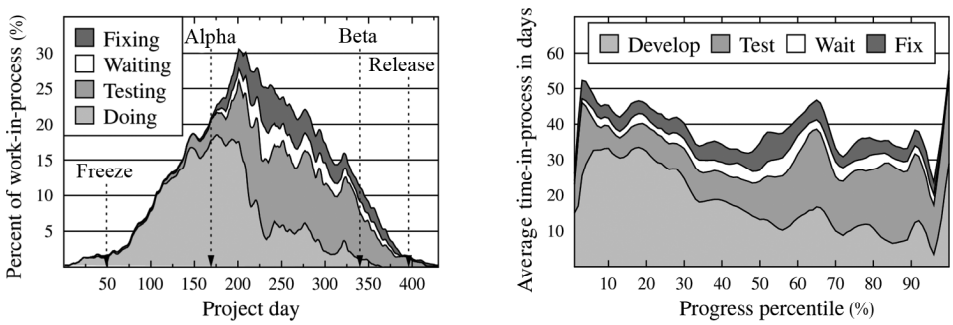

Figure 4. Little's Law work-in-process (left) and time-in-process (right) graphs.

4.2 Cumulative Progress

Figure 3 shows progress "S"-curves for various steps in the process as found in the repository logs. We measure progress by the number of tasks started or completed. A project is 50% complete when 50% of the tasks have been completed. In the graphs, the horizontal dimension is days into the project, while the vertical dimension represents percent of progress by a given day. Going from left to right in the graph on the left, the curves show the day by which given percentages of requirement tasks were started and when requirements tasks were completed. In the graph on the right, the curves from left to right are our estimate of when defects were caused, when defects were reported, when defect repair tasks were started, and when defect repair tasks were completed.

The slope of the curves reflects the rate at which tasks were started or completed. Significant testing and repair does not start until alpha. The process batches system testing until alpha, so testers will have a significant amount of code before starting tests. Waiting for a batch to accumulate is an obvious bottleneck. While some defects were found within days of alpha, the slope of defects found rises sharply only 6 weeks later. In talking with testers on the project, we learned that while they had the requirements all along, they first saw the code at alpha, a handover, and then took several weeks to figure out how to apply the bulk of their tests to the actual code – an example of motion.

The horizontal distance between curves is a good indicator of the time needed to go from one stage to the next. We were able to confirm this approximation by comparing the distances with the actual per task data. Per-task time between starting and completing a repair was typically 1–3 days.

While much can be said about the relative and absolute durations of the steps, one detail in particular caught our attention: the time between finding a defect and starting a repair. The project maintains a constant inventory of defects waiting to be fixed. In Fig. 3, distances on the horizontal axis show this time. Looking at the vertical distances, which represent amounts of work, the ratio between bugs in queue and bugs in repair seems oddly consistent. We discuss the interpretation of distance on both axes using Little's Law, below.

4.3 Little's Law

From queuing theory, Little's Law tells us that the vertical, distance between progress curves is a good measure of the amount of partially completed work, called work-in-process. The horizontal distances show process durations, called time-in-process [11]. From the data in Fig. 3 it is easy to plot both of the Little's Law views. Figure 4 presents the results. The bottom to top order for both graphs is: requirements work, testing or waiting to be tested, waiting to be repaired, and repair work.

From a Lean perspective, the danger of having lots of work in process is that it causes instability when requirements change – a phenomenon we have in fact seen on other projects. It is hard to complete tasks whose requirements or whose start and end points are no longer valid, and hard to start new tasks when their relationship to the current state is unclear. Until this problem is

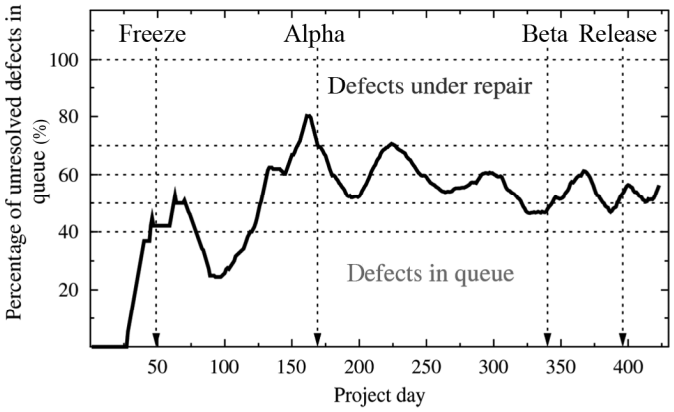

Figure 5. Time series control chart of defect queue size as proportion of unresolved defects.

resolved, the system is unstable. Reducing the amount of work-in-process requires reducing per-task durations.

The time-in-process graph shows the average time it takes a work task to make it through the process, or cycle time, at different stages of progress. Time-in-process includes not just the time of the requirements task, but also the time for testing and resolving defects. The overall average in-process time is 40 days. Again, we were able to confirm these figures with the actual per-task measurements. Testing time seems to increase later in the project. The increase is more likely an artifact of our conservative estimate becoming more accurate as the backlog declines. The sharp spike at the very end reflects the completion of a few low priority tasks that lingered for excessively long time.

5. Focused Analysis

The wealth of data found in the repository supports a wide range of analysis techniques. Using this data, we can ask specific questions, view relationships, and test hypotheses. In a focused analysis phase we can look deeper into a specific issue, and pursue it from a variety of angles. To illustrate the possibilities, we present three very different techniques: a two-factor time series analysis, a simulation, and a confidence test for correlation. We use all three to address questions about the defect queue.

5.1 Time Series

The initial analysis raised questions about the size of the defect queue and its apparent relationship to the number of repair tasks being worked on. Figure 5 shows a time series of the size of the defect queue as a proportion of unresolved defect tasks.

Few defects were reported or worked on prior to day 155, as shown in Fig. 2 and the right side of Fig. 3. Figure 5 shows an abrupt end to the rising relative queue size on day 155. After day 200, the size of the queue stays between 50 and 60% of unresolved defects. This period coincides with the period of developers being released from the project in Fig. 2. The queue is managed with the developer pool.

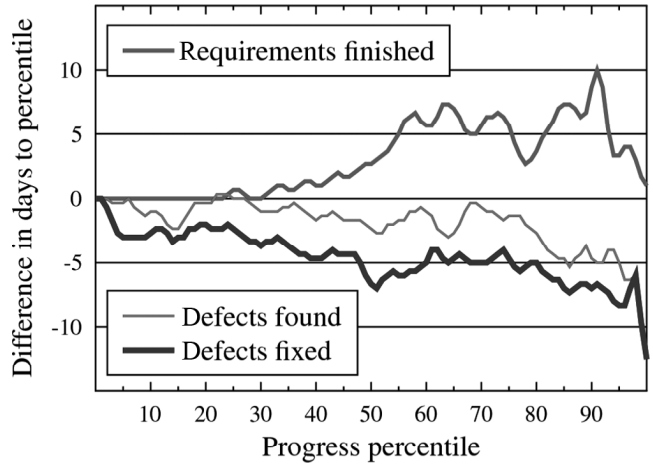

Figure 6. Comparison of two simulations with different defect queue policies.

5.2 Simulation

While defects sit in queue, new requirements tasks continue to be started. Testing time is needed to find defects. System testing, and getting features to test, has a high priority in waterfall development. In Lean and Agile practices, repairs have the highest priority. One hypothesis might be that giving defect repair the highest priority would lead to later starts for feature testing, and as a result delay project completion. Using project data, we can test the hypothesis that giving feature development more priority than repairs leads to an earlier overall completion.

To test the hypothesis, we ran two simulations using actual project events as input. In both simulations we used the same tasks executed in the same order and requiring the same effort to complete. The same number of developers became available on the same days, and the same number of developers became eligible for release on the same days, thus maintaining the same level of effort. Using the conservative estimate of the causes, defects were given the same number of days in test after the last known change entered the system.

With the queue managed as in Fig. 5, the behaviour of the simulation was very close to what was actually observed. The completion day of the simulation was the same as in the actual project. This result provided confirmation that the simulation was a reasonable representation of the actual process.

In the second, test simulation, repairs got priority when new tasks were assigned. Everything else was kept the same. The difference between the two trajectories, measured in work days, appears in Fig. 6. Weekends were excluded from the difference. The sudden variation at the very end of the graph in Fig. 6 reflects the same long running tasks visible at the end of Fig. 4. Because of the change in queue-policy, they are handled earlier in the second simulation.

The time needed to reach first alpha levels of completion was the same in both simulations. As expected, defects traced directly to requirements were found slightly later in

Table 1
Work Effort in Days versus Age in Calendar Days of Defects at the Time of Work Started, Computed for a Larger Project

Age Range (Cause to Start) in Days	Average Repair Effort in Days	Number of Tasks
0–20	7.1612	1,737
20–40	9.5407	627
40–60	12.2181	298
60–80	11.6095	169
80–100	15.6640	125
100–120	18.6813	91
120–140	20.6667	75

the test case. But the average time of finding defects was earlier. With the same numbers of developers, and only a change in queue policy, the second simulation reached completion 2 weeks earlier. The data does not support the hypothesis that maintaining a defect queue produces an earlier project completion date than fixing defects when they are found. It shows the opposite – potentially 17 calendar days or 5% of total.

In the development process, repairs may cause, and also uncover, more defects. So other defects are found and the cycle repeats. Defect tasks are more numerous than requirements, and tend to be clustered around hot spots in development. Getting the repairs fixed and back into testing is as important, and in the long run more important, than getting the first tests started early. In Lean terminology, the repair cycle is a feedback loop. Keeping repairs in queue makes the cycle longer. Rerunning the Section 4.3 analysis on the test data shows that simply ending the backlog practice could reduce the average cycle time from 40 days to 37 days. To test for bias, we tried using different causality models and omitting especially high defect prone items (outliers) from the data. But the result did not change.

5.3 Correlation

Leaving defects in the queue has another cost. As the defect sits in queue, both the context of its creation and the tests that found it become more remote and increasingly difficult to reconstruct. This corresponds to the inventory waste in Lean analysis. We tested a second hypothesis, that the per-repair cost of fixing defects increases with the time between when a defect is caused and when it is repaired.

Table 1 shows a correlation table using data on 3,000 defect repairs from a different, larger project. The data is grouped by the time between our conservative estimate of the cause and the start of the work. The data was grouped by age in increments of 20 days. The second column shows that average repair time, measured as the difference between its first and last configuration change. A least squares line fitted to all 3,000 data points had a slope of 0.108, which would mean 1 day of extra effort for every 10 days of wait. The correlation's R value was 0.20. The F statistic confidence level for rejecting the null hypothesis was better than 0.01, thus supporting the hypothesis.

The same analysis on data from the project being analysed here was less dramatic, but still significant. The correlation line had a slope of 0.05 and an R value of 0.10. But the F statistic was still significant at the 0.01 level. The hypothesis is supported. Adding waiting time to the queue adds effort to the eventual repair. Using the 0.05 line, the cost to this project would exceed 150 task hours.

5.4 Measurement

A third hypothesis about the defect inventory is that while defects wait in queues developers are implementing new requirements on top of known, or possibly known, defects. In this case, the test is to find the proportion of tasks that are started with an artifact that, at the time it was checked out, had a defect that had not been fixed. After removing outliers, of the 280 requirements tasks started, 90 involved files that were later edited for a defect that had already been submitted – 32% of requirements were built on files with known defects. Using the conservative estimate for time of defect creation, the proportion of tasks starting with buggy code is at least 43%. We can compare these numbers to the simulation in the previous section, where repairs were given higher priority. In the simulated process, only 23% of the requirements tasks had known prior defects, while a minimum of 38% started with buggy code – a sizeable reduction.

The observations discussed here underscored the strong commitment to testing already present in the organization. But they also revealed a need for greater emphasis on repair. The improvement can be incremental. Our analysis highlights the issue with a wide range of metrics that can be used to track the improvement as it evolves.

6. Related Work

In surveys of the literature, by Niazi *et al.* [12], Rainer and Hall [13], and others, the emphasis seems always to be on process adoption (ala CMMI or ISO 15504) and work estimation. There is little mention of improving business performance. Niazi goes so far as to say that the current problem is a lack of effective strategies to implement standards and models [12]. Process metrics are frequently mentioned as a goal. Metrics, like questionnaires, are used to improve compliance, but not change the process. It is interesting to note that where graphs of progress and effort have appeared, the goal is always estimation, not process change. For example, graphs of effort appear in both Mockus *et al.* [9] and Donzelli [14]. A graph of progress appears in Kan *et al.* [8]. The graphs in both Mockus and Kan showed much higher rates of defect effort than the project shown here. None of these papers mentioned the possibility that the process itself could be improved.

Boehm and Turner [15] list traditional engineering measurements as a significant barrier to implementing agile processes. Their conclusion identifies the need for metrics data to validate value and capture lessons learned as "most important." But no metrics are given. Hartman and Daymond [16] provide guidelines and criteria for process metrics emphasizing business value for use with Agile and Lean methodologies. Similarly, Lindvall *et al.* [17] list five different value levels for software measurement: characterize, understand, evaluate, predict, and improve, and describe a process for breaking goals into smaller questions. Again no actual metrics are described. Our approach satisfies all of these criteria.

There is considerable literature on the use of repository data. Kim and Whitehead [18] used repository data to look at defect fix times, using data from an open source project. Average fix times were 200 days, or an order of magnitude greater than here; the issues are clearly different. Sliwerski *et al.* [19] use line-by-line source code analysis to map defects to causes. Their approach is more accurate than ours, but requires considerably more effort. For the analysis, we need significance more than precision. Lower bound values are sufficient for testing a null hypothesis. Fischer *et al.* [20] and Kim *et al.* [21] report work on tools for extracting data from repositories, largely for code and architecture analysis. Architectural analysis is a useful complement to the work presented here. Both groups report novel, but not reliable, solutions to the correlation problem. Our data is less problematic, due to mature practices.

Li *et al.* [22] present recent industry work not unlike our own. Their approach looks for variations outside preset thresholds, while ours look for bottlenecks and delays with no preset threshold. Interestingly, they also found excessive delays in starting bug repairs. In their case, the delay was traced to testers batching preliminary test results for a subsequent confirmation step.

The work presented here continues our pursuit of adoption-centric improvement [23]. Earlier we presented the idea of using repository data in a Six Sigma initiative [1]. Six Sigma focuses on the measurement of intermediate factors and a search for causes of variation. More recently, we presented a subset of these graphs without the analysis [24], and a parallel use of the data to investigate code and architecture [25].

7. Conclusion

In this article we present a new approach to process improvement based on empirical data and sound analysis practices. The approach focuses on finding waste in process practices. We use data collected non-invasively and at minimal cost. In mature organizations, the data should already exist. Through a variety of models presented in the examples, we show how to find indicators of various types of waste, and provide baseline measurements against which to make comparisons when improvements to the process are applied, or simulated. By comparison, traditional approaches to process improvement focus on process adoption and the behaviour of individuals. Metrics that support traditional initiatives are costly to collect and unreliable.

In a focused analysis phase, we looked at a particular issue in the project being investigated. Through several different forms of analysis, we showed how a simple change in the task scheduling policy could reduce time to market by several weeks, or more than 5% of overall development time. Every organization will have its own processes and issues. Scheduling defect tasks is just one issue. Following the approach of investigation shown here, and using their own repository numbers, organizations of all kinds should be able to better understand their processes and investigate waste.

In this article, we present analyses based only on configuration and task tracking data. A full repository should also include requirements and testing data. Such data could be used for additional and equally significant analyses.

References

[1] M. VanHilst, P. Garg, & C. Lo, Repository mining and Six Sigma for process improvement, *Proc. International Workshop on Mining Software Repositories*, 2005.

[2] J.M. Morgan & J.F. Liker, *The Toyota product development system: integrating people, process and technology* (New York, NY: Productivity Press, 2006).

[3] M.N. Kennedy, *Product development for the Lean enterprise: why Toyota's system is four times more productive and how you can implement it* (Richmond, VA: Oaklea Press, 2003).

[4] *The lean aerospace initiative*, Available: http://lean.mit.edu. Last accessed 19 June 2009.

[5] M. Poppendieck & T. Poppendieck, *Lean software development: an agile toolkit* (Upper Saddle River, NJ: Addison-Wesley, 2003).

[6] M. Poppendieck & T. Poppendiech, *Implementing Lean software development: from concept to cash* (Upper Saddle River, NJ: Addison-Wesley, 2007).

[7] S. Shingo, *A study of the Toyota production system* (New York, NY: Productivity Press, 1989).

[8] S.H. Kan, J. Parrish, & D. Manlove, In process metrics for software testing, *IBM Systems Journal*, *40*(1), 2001, 220–241.

[9] A. Mockus, D. Weiss, & P. Zhang, Understanding and predicting effort in software projects, *Proc. 25th Int. Conf. on Software Engineering*, 2003, 274–284.

[10] P. Middleton & J. Sutton, *Lean software strategies: proven techniques for managers and developers* (New York, NY: Productivity Press, 2005).

[11] D.G. Reinertsen, *Managing the design factory: a product developer's toolkit* (New York, NY: The Free Press, 1997).

[12] M. Niazi, D. Wilson, & D. Zowghi, A maturity model for the implementation of software process improvement: an empirical study, *Journal of Systems and Software*, *74*, 2005, 155–172.

[13] A. Rainer & T. Hall, A quantitative and qualitative analysis of factors affecting software processes, *Journal of Systems and Software*, *66*, 2003, 7–21.

[14] P. Donzelli, A decision support systems for software project management, *IEEE Software*, *23*(4), 2006, 67–75.

[15] B. Boehm & M. Turner, Management challenges to implementing agile processes in traditional development organizations, *IEEE Software*, *22*(5), 2005, 30–39.

[16] D. Hartman & R. Daymond, Appropriate agile measurement: using metrics and diagnostics to deliver business value, *Proc. IEEE AGILE 2006 Conf.*, 2006.

[17] M. Lindvall, P. Donzelli, S. Asgari, & V. Basili, Towards reusable measurement patterns, *Proc. 11th IEEE Int. Software Metrics Symposium*, 2005, 21–28.

[18] S. Kim & E.J. Whitehead, Jr., How long did it take to fix bugs?, *Proc. Int. Workshop on Mining Software Repositories*, 2006, 173–174.

[19] J. Sliwerski, T. Zimmerman, & A. Zeller, When do changes induce fixes?, *Proc. Int. Workshop on Mining Software Repositories*, 2005.

[20] M. Fischer, M. Pinzger, & H. Gall, Populating a release history database from version control and bug tracking systems, *Proc. Int. Conf. on Software Maintenance*, 2003, 23–33.

[21] S. Kim, T. Zimmermann, M. Kim, A. Hassan, A. Mockus, T. Girba, M. Pinzger, E.J. Whitehead, Jr., & A. Zeller, TA-RE: an exchange language for mining software repositories, *Proc. Int. Workshop on Mining Software Repositories*, 2006, 22–25.

[22] J. Li, N. Jiang, Q. Wang, & Y. Yang, Tracking projects through a three-dimensional software development model, *Proc. 31st Annual Int. Computer Software and Applications Conf.*, 2007, v.1, 301–308.

[23] S. Huang, S.R. Tilley, M. VanHilst, & D. Distante, Adoption-centric software maintenance process improvement via information integration, *Proc. 13th IEEE Int. Workshop on Software Technology and Engineering Practice*, 2005, 25–34.

[24] M. VanHilst & S. Huang, Mining objective process metrics from repository data, *Proc. 19th IEEE Int. Conf. on Software Engineering and Knowledge Engineering*, 2009.

[25] S. Huang & C. Lo, Analyzing configuration management repository data for software process improvement, *Proc. 19th IEEE Int. Conf. on Software Engineering and Knowledge Engineering*, 2007.

Biographies

Michael VanHilst is a research professor in the College of Engineering and Computer Science at Florida Atlantic University. His research interests include software development methods, software process improvement, and computer security. He has 20 years of industry experience. He received his Ph.D. degree in computer science from the University of Washington. He is a member of the IEEE Computer Society and a senior member of the ACM.

Shihong Huang is an assistant professor in the Department of Computer Science and Engineering at the Florida Atlantic University. Her research interests include reverse engineering, program comprehension, and software maintenance and evolution. She was the General Chair of the 24th ACM International Conference on Design of Communication, and Program Co-Chair of the 9th IEEE International Symposium on Web Site Evolution. He received her Ph.D. in computer science from the University of California, Riverside. She is a member of the IEEE Computer Society and the ACM.

Hugh Lindsay is a software engineer principle at General Dynamics C4 Systems. Earlier he was a distinguished member of the technical staff (R&D) at Motorola Mobile Devices where his responsibilities included development tools and Six Sigma initiatives.