Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering
(SEKE'2009: July 1 – 3, 2009; Boston, MA)

Mining Objective Process Metrics from Repository Data

Michael VanHilst, Shihong Huang
Florida Atlantic University
777 Glades Road
Boca Raton, Florida
(mike, shihong)@cse.fau.edu

Abstract

The configuration management repository includes abundant data not only on configuration items, but about the process itself. But meaningful information about the software process is often hidden. This paper presents a method of extracting software process metrics from software repositories. More specifically, the metrics presented use data from the bug or task tracker and from the configuration management event log. The metrics are presented in graphic forms common to traditional and lean project management practices. The metrics presented here are empirical – not subject to bias in reporting or interpretation, and focused on measuring the process itself - not the developers and artifacts. They are derived from data that commonly exist in project software repositories, and thus can be collected with little or no cost. The metrics are illustrated with real software development repository data collected from a large industry project over a time span of several years.

Keywords: software process improvement, process metrics, information visualization, data analysis, and software repository

I. INTRODUCTION

Today, many software organizations make disciplined use of artifact management tools. These tools and practices create a unique source of development metrics. The metrics are objective – tied to actual process events, and non-invasive – extracted without additional developer input or effort [1]. The likelihood that any organization that is interested in metrics already uses tools and disciplined practices is high. In this paper we present a collection of software process metrics using only the event logs from two common development tools: bug/task tracking and configuration management.

This paper focuses on the collection and visualization of metrics that characterize the process itself. Our original interest was motivated by recent literature on process improvement. Common notions of process improvement address uncertainty and failure, and focus on conformance to best practices and improving estimates of time and effort. This earlier work gives us a range of metrics to measure the complexity of code and the behavior of developers. More recent literature on process improvement, has been motivated by Agile and Lean practices and the application of Six Sigma to software It shifts the focus to reducing cost and improving time to market. Improvement requires changes to the process. The measurements we present here support such change with baseline and comparison metrics. The metrics we use for progress, throughput time, and effort are not new. But the approach of deriving them non-invasively and at low cost from repository event logs is new.

We validated our approach on data collected from several large multi-month commercial projects involving hundreds of developers, thousands of artifacts, and tens of thousands of events. We illustrate the examples in this paper with graphs produced from one of those project.

Section II presents motivation and related work, Section III explains how the data is extracted from the repository and regrouped for analysis. Section IV presents several different graphical views of the process. Section V concludes with a brief comment on future work.

II. MOTIVATION AND RELATED WORK

Often metrics that are described as 'process metrics' actually measure individual developers or properties of specific artifacts [2]. This is especially the case where software process improvement is viewed as improving conformance to existing practice. By measuring only how well developers are working, regardless of the problem being defined, the solution is for developers to work harder or with greater care. Our developers are already working overtime. We wanted to complement the data with measurements of how the process itself is working, independent of developer behavior.

We feel that with today's competing development methodologies, there is more need for more metrics, and a greater variety of metrics, than ever before. Boehm and Turner, for example, list traditional engineering measurements as a significant barrier to implementing agile processes [3]. Their conclusion identifies the need for metrics data to validate value and capture lessons learned as "most important." But no specifics are given.

[4][5] and others have surveyed the literature on software process improvement. The emphasis seems always to be on process adoption with little mention of improving business performance. [4] goes so far as to say that the current problem is a lack of effective strategies to implement process standards and models.

Process metrics that do exist often use questionnaires. The data is necessarily subjective and costly to collect. Overheads range from tens of person hours for CMMI Class C or ISO 15504 appraisals, to hundreds of person hours for a full CMMI Class A appraisal [6][7]. A supposedly light weight software process assessment describe in [8], recently spent 979 and 1376 hours in interviews.

[9] presents measurement models that are perhaps the closest to those presented here. But the 'repository' data is actually from manual reporting tools and is used not to assess the process, but how well individuals conform to it.

Hartmann and Dymond [10] provide guidelines and criteria for metrics emphasizing business value for use with agile and lean methodologies. But no actual metrics are described. The metrics presented here are consistent with those criteria.

The Poppendiecks [11] include measurement models, such as value stream maps and pareto charts, in their discussion of lean software development. But they don't discuss how to collect the data. Their mention of Little's Law, to estimate queue waiting and completion times, inspired us to investigate its use further. Little's Law is described with more detail in [12].

Johnson et al. describe collecting data automatically from sensors in various development tools [13]. While this work complements our own (i.e. Johnson's daily build metrics). But, like the others, they focus either on efficiency as a characteristic of individuals or improving the accuracy of predictions within a given process. Work with repositories often use data from open source project. Results from such data may not apply to industry practices. Fix times in [14], for example, were 200 days. Fix times in our data are typically less than 5 days.

Earlier tools designed to collect data from integrated development environments, such as TAME [15], AMADEUS [16], and APSE [17], assumed the existence of a process model. They would then measure and analyze how well the process plan was being executed. In practice, this approach yielded little value and eventually died. We don't assume a process model. The model is revealed in the data. Ultimately, we are not as interested measuring plan execution as assessing process design.

III. DATA COLLECTION AND REDUCTION

A software development project is made up of work pieces called tasks. Tasks divide a large project into small manageable units of work. Tasks are assigned to specific developers, they trace specific work to a specific deliverable, and their completion represents tangible progress. A task can represent a changed or added piece of functionality, or a defect to repair. While useful as a management unit, different tasks can differ in both effort consumed and value produced.

A task begins with an initial request and ends with final integration into the product. As they make their way through the development process, tasks are under active development and undergo testing. They also often wait in queues. We can track their progress.

In a disciplined development organization, every code change is associated with an identified task. Tasks may be requests to implement or change a feature, or defect reports needing resolution. Both kinds of tasks are tracked with task or bug tracking tools, for example Bugzilla or ClearQuest. A task tracker records the submission date of the request or defect report, the date work began, and the date of completion. Other information may also be recorded, such as estimated hours of effort. But for our purposes, we only use the task ID, the task classification (new requirement or repair) and the date of submission. As discussed below, we prefer to use other, more objective sources for the remaining dates.

Code development in a disciplined development organization depends on configuration management. No code change occurs without artifacts being checked in to the configuration management system. Each check-in event identifies an artifact, the date and time, and a developer. When a developer starts work on a new task, he or she checks in artifacts associated with that task. As work progresses, more check-ins occur. When work is completed, no more artifacts are checked in for that task. In order to correlate events, and for us to reconstruct the development history, the event record must include a reference to the associated task.

While some configuration management tools do not require, or directly support task ID's, task correlation is important to requirements traceability and should be common practice. The ID can be included within the comment, or within the name of a branch. In the latter case, all task work occurs in its own branch. In the future, we expect configuration management tools to better support correlation with tasks.

To produce process metrics, we extract the task tracker and configuration management event logs, clean them to correct misspellings and remove unimportant events and information, and deposit the result into a database. We then construct a table of artifacts and tasks, where each record represents a single artifact's involvement in a single task. The record's fields include the artifact and task identifiers, and the dates of first and last change. We find task begin and end dates by grouping all records by task ID, and taking the dates of earliest and latest change. These dates are combined with the task tracker submission date to get a complete history.

For defect repair tasks, we are also interested in the date when the defect was created. To assign a date, we use the following assumption: the defect was created by a change to an artifact that was later edited to correct the defect. Using this assumption, for each artifact involved in the repair task, we look at all prior changes for all other tasks that were completed prior to the defect's submission date. Taking a conservative approach, we choose the latest date from the set of candidate changes – the date nearest to the defect report. This approach could be refined with additional information, such as line numbers involved. But for our purposes, this simple conservative approach produces a usable result.

Using this repository data, for each repair task, we know how long it took between when work first began and work finally ended, we know how long the task spent in the defect queue between the time it was reported and work first began, and we infer how long it took between the time the defect was caused and when it was found,

Since we know the events we used as defect causes, we can associate defects with prior tasks. For a given feature task, we not only know the time it took to complete the task, we also have the defects that were caused, how long it took to resolve those defects, and even dates of later defects created in the first round of defect repair work.

We measure effort in developer days. A developer is assumed to be active on a task on every day between the first and last change events that associate 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 features if they are active on a request task, and they are counted as actively repairing defects if they are active on a repair task. It is possible that the same developer is active on two or more tasks on the same day, and even on one task of each kind on the same day. Our data takes this into account. On the projects for which we have collected data, both situations do occur, but not very often.

IV. VISUALIZATION

A. Effort.

Figure 1 shows an idealized graph of effort over time. In an ideal world, a team of programmers is assigned to a project and starts working as soon as requirements are available. Work then continues at a steady pace until all of the requirements are completed, with minimal taper at the beginning and the end. The graph would essentially have a boxcar shape. Real projects, of course, do not look like the graph in Figure 1. They look, instead, more like the graph in Figure 2. Developers must be freed from other projects or hired new at the beginning. Some tasks are likely to drag out at the end, keeping smaller numbers of developers busy beyond the end. Defects are found as the product takes shape and new effort must be devoted to fixing these defects. Testing itself takes time. Some defects require more testing than others before they are found. As depicted in Figure 2, effort ramps up as developers become available and peaks somewhere in the middle. A second later hump of effort addresses rework after tests. The overall effort may not have as pronounced a second hump as depicted in Figure 2, since the later efforts reflect both the rework and requirements that either dragged on or were started late.

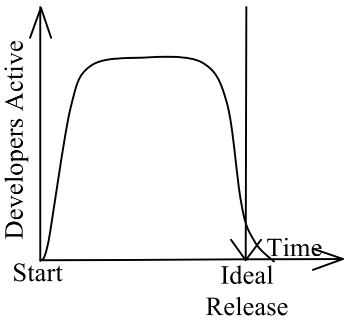

Figure 1. Projects effort vs. time in an ideal process

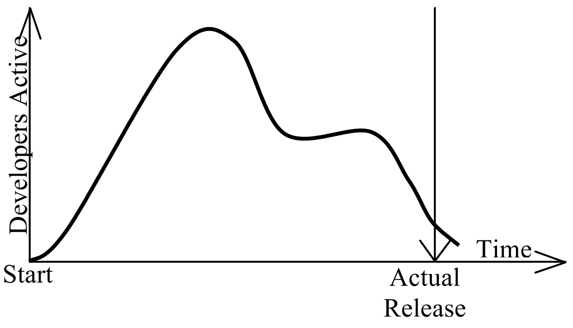

Figure 2. Project effort vs. time with repairs and delays.

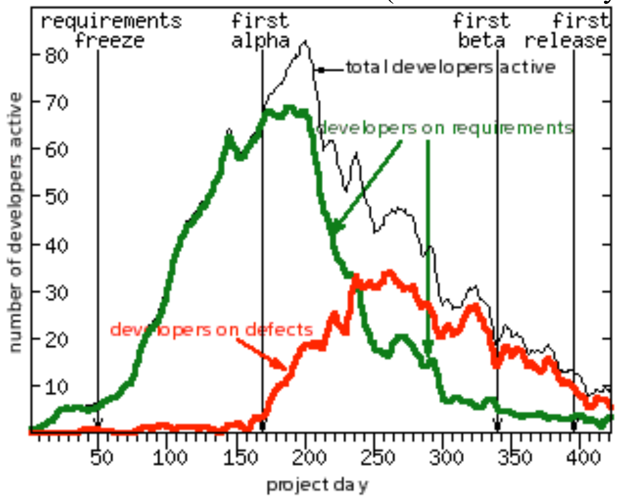

Figure 3. Effort vs. calendar day from real data

Figure 3 shows the real graph of effort vs. time for a project that lasted more than a year.. The vertical axis shows the number of developers active on any given day. The horizontal axis shows the day, counted in number of days from the beginning of the project. The project followed a waterfall process; we indicate the dates of the waterfall milestones for requirements freeze, alpha testing release, beta release, and first customer release. The thin line on top shows overall effort, while the two thicker lines show effort separately for requirements and repairs.

Total project effort, measured in developer days, is the area under the curve. We compute it by summing the developers active on each day. The X axis in Figure 3 is actually calendar day. We exclude weekends and holidays from the sum. While total effort differs from one project to the next, the ratio between requirements effort and repair effort is more constant and can be used as a baseline for comparison in process improvement. In this project, 33% of the total effort is devoted to repair. In comparison with other waterfall projects in published literature, and our own experience, 33% is common, and actually better than average.

At a more detailed level, we know that developers do more than code on any given day. But in aggregation, differences in activity pattern of one developer from one day to the next average out and appear the same from one project to the next.

B. Cumulative Progress

For measuring progress, we use task as a unit of progress. A project is 50% complete when 50% of the tasks have been completed. We use 5 separate metrics for measuring progress: number of requirements tasks begun, number of requirements tasks completed, number of defects found, number of repair tasks begun, and number of repair tasks completed. From the dates we derived for when each defect was caused, described earlier, we can construct a sixth progress metric: number of defect created.

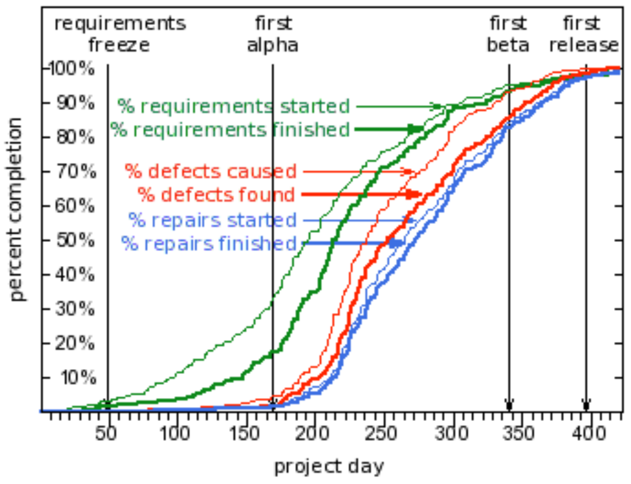

Figure 4. Progress curves from real data

We can plot proportional completion by dividing the number of tasks completed by the total number of tasks in the project. We plotted all 6 curves on the same graph in Figure 4. From left to right the curves are percent of requirements tasks begun, percent of requirements tasks completed, percent of defects created, percent of defects found, percent or repairs begun, and percent of repairs completed. The plots are classic S-curves.

Our conservative estimate of the defect creation events probably puts the "defects caused" curve to the right of where it should be. The left two curves in Figure 4 are measured in terms of requirements tasks, while the remaining four curves on the right are measured in terms of repair tasks. We must be careful not to give meaning to the distance between the two groups – between the 2nd and 3rd curves. They may in fact cross.

The slope of a curve indicates the rate of progress at that point in time. Variations can indicate events or problems. We don't see a significant uptick in defects being found until day 205 – 4 weeks after alpha testing supposedly began. .A significant number of requirement tasks were completed around day 210, which corresponds to a time of developers being released in Figure 3.

C. Little's Law

From queuing theory, Little's Law tells us that the vertical distance between two curves in the progress graph is a good indicator of the amount of work, or in our case the number of tasks, currently in that phase of the process. This measure is called work-in-process. The vertical distance between the 'defects found' curve and the 'repairs started' curve is an indicator of the number of defect tasks waiting in the defect queue. The vertical distance between the 'defects caused' curve and the 'defects found' curve is an indicator of the number of defects in testing waiting to be found.

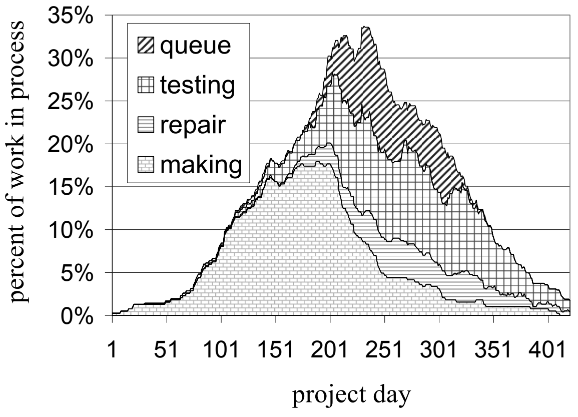

Figure 5. Little's Law work in process

Figure 5 shows the Little's Law distances for work-In-process plotted in Excel as an area chart. The area at the bottom is work on requirements, which is distance from the requirements 'requirements started' curve to the 'requirements finished' curve. Above that we show work on repairs, computed as the distance from the 'repairs started' curve to the 'repairs finished' curve. Above the repair work we show the defects undergoing test, from the distance between the 'defects caused' and 'defects found' curves. The area at the very top is for defect reports sitting in queue, measured between the 'defects found' curve and 'repairs begun.' The order, bottom to top, was chosen to put the most productive activities on the bottom and the least productive phase at the top.

In Lean and Agile practices, it is considered good to minimize the amount of work in process at any given time. It is hard to make changes when much of the work is midstream in process. A project will be more agile if there is less work in process to better accomodate change.

Little's Law also tells us that the horizontal distance between two curves in the process graph is a good indicator of the amount of time it takes a piece of work to make it through phases of the process. This measure is called time-in-process. The horizontal distance between the defects found curve and the repairs started curve tells us defect reports commonly spend 5 to 10 days in the defect queue. Since we have the actual dates for every repair task, we were able to confirm that that is true.

Figure 6 show the Excel area chart for Little's Law time in process. It is common for work that takes the longest to be completed at the end.

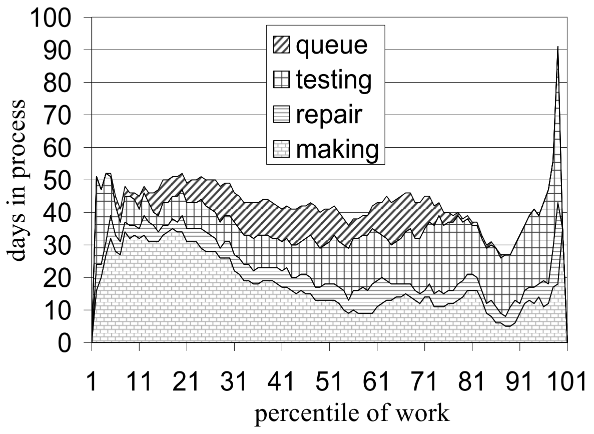

Figure 6. Little's Law time in process.

By Little's Law, the entire distance from the top to the bottom is the time it takes a piece of work to make it all the way through the process, from when work initially started to when the defects are repaired and it is ready to be in the finished product. This measure is called throughput time. The shorter the throughput time, the more responsive the process can be to new or changing demands.

From the repository event data, not only were we able to extract dates for every requirement task and defect repair, but also to link downstream defects to prior tasks. For a given requirement task, we can assign the repairs it took to get it right. Thus we have real process time for every requirement request. Since there is little uniformity among tasks, the real plot is a lot noisier than that produced by Little's Law. Because the data is from an industry project and contains proprietary information, we cannot publish the real per-task graph. But the average thoughput time, computed directly from the per-task data, was 45 days. This value is consistent with the Little's Law time in process graph.

D. Progress Path

Vanderwall recently presented a graph of progress that he calls the Project Progress Viewer [18]. The PPV plots completed functionality, on the X axis, against tests passed, on the Y axis. The resulting graph shows the process' path history. The points along the path have time values which also give velocity, and can be used to project future progress.

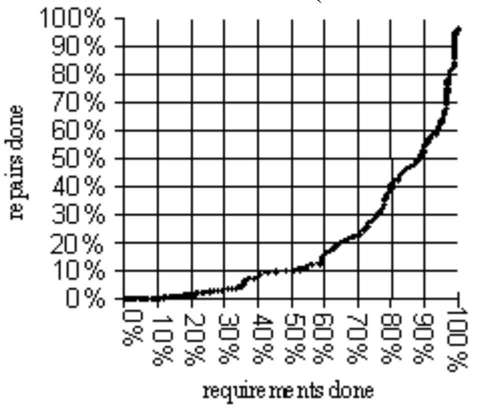

Figure 7. Progress path of requirements vs. repairs.

The shape of the curve in Figure 7 is classic waterfall. Requirements make early progress while defect repair lags significantly behind. In a Lean or Agile process, the graph will present more of a straight line. The project here made good mid-period progress, but slowed considerably towards the end. Not that if the project isn't done, actual numbers can be used in place of percentages, as Vanderwall does.

V. Conclusion and Future Work

In this paper we presented a method of extracting process metrics from common repository data. The data was collected without questionnaires, without adding instrumentation to tools, and without asking developers to provide any information other than what is already provided as part of normal practice. We also showed six basic process analysis graphs that can be drawn from that data. Our intention is to show how objective process metrics can easily be obtained and to draw attention to the opportunity to analyze the process itself. An actual analysis will depend on the goals of the organization and the types of improvements being considered. We leave that discussion for another paper.

We have done in-depth analyses of several processes including the one shown here. Additional graphs and numerical analyses, beyond those shown here, were used in the analyses. Some details revealed and further investigated in those analyses are visible in the graphs shown here, but are not discussed. We hope to publish that work soon.

The work presented here was started for the purpose of doing Six Sigma analyses for process improvement[19]. We also have related work on the role of artifact and architecture hotspots in process issues, using the same repository data [20].

Bibliography

[1] S. Huang, S. R. Tilley, M. VanHilst and D. Distante, Adoption-centric software maintenance process improvement via information integration. 13th *IEEE International Workshop on Software Technology and Engineering Practice* (STEP 2005), 2005, pp. 25-34

[2] R. Dumke, E. Foltin, R. Koeppe, and A. Winkler, *Softwarequalität durch Meßtools: Assessment, Messung und instrumentierte ISO 9000*. Vieweg Press, 1998.

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

[4] M. Niazi, D. Wilson, and D. Zowghi. A maturity model for the implementation of software process improvement: an empirical study. *Journal of Systems and Software*, 74(1) January 2005 pp. 155-172.

[5] A. Rainer and T. Hall, A quantitative and qualitative analysis of factors affecting software processes, *Journal of Systems and Software*, 66(1) April 2003,pp. 7-21.

[6] S. Zahran. *Software process improvement: practical guidelines for business success*. Addison-Wesley. 1998.

[7] K. El Emam and L. Briand. Costs and benefits of software process improvement. Technical Report ISERN 97-12, Fraunhofer Institute for Experimental Software Engineering Engineering, 1997

[8] F. Pettersson, M. Ivarsson, T. Gorschek, and P. Őhman. A practitioner's guide to light weight software process assessment and improvement planning. *The Journal of Systems and Software*. 81(6) June 2008, pp. 972-995.

[9] Y. Zhang and D. Sheth. Mining software repositories for model-driven development. *IEEE Software*, (23)1:82-90.

[10] D. Hartman and R. Daymond. Appropriate agile measurement: Using metrics and diagnostics to deliver business value. *IEEE Agile Conference*, 2006, pp. 126-134.

[11] M. Poppendieck and T. Poppendiech. *Implementing lean software development: From concept to cash*. Addison-Wesley, 2007.

[12] D.G. Reinertsen. *Managing the design factory: A product developer's toolkit*. The Free Press, 1997.

[13] P. Johnson, H. Kou, M. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita. Improving software development management through software project telemetry, *IEEE Software,* 22(4) July 2005, pp. 76-85.

[14] S. Kim and E.J. Whitehead, Jr. How long did it take to fix bugs? *International Workshop on Mining Software Repositories*, 2006, pp. 173-174

[15] V. R. Basili and H. D. Rombach. The TAME project: Towards improvementoriented software environments. *IEEE Transactions on Software Engineering*, 14(6, June 1988, pp. 758-773.

[16] R. W. Selby, A. A. Porter, D. C. Schmidt, and J. Berney. Metric-driven analysis and feedback systems for enabling empirically guided software development. *International Conference on Software Engineering*, 1991, pp. 288-298.

[17] The Perfect Consortium. APEL abstract process engine language. Perfect Handbook Booklet, European Esprit Project, 1997.

[18] R. Vanderwall. The chart that saved the world. *Software Test and Performance*. (6)1 January 2009, pp. 8-1.

[19] M. VanHilst, P.K. Garg, and C. Lo. Repository mining and Six Sigma for process improvement. *International Workshop on Mining Software Repositories,* 2005, pp. 1-4.

[20] S. Huang and C. Lo. Analyzing configuration management repository data for software process improvement, *IEEE International Conference on Software Engineering and Knowledge Engineering* (SEKE) 2007.