Measuring Effort in a Corporate Repository

Michael VanHilst, Shihong Huang, James Mulcahy
College of Computer Science & Engineering
Florida Atlantic University
Boca Raton, Florida, United States
(vanhilst, shihong, jmulcah1)@fau.edu

Wayne Ballantyne, Ed Suarez-Rivero,
Douglas Harwood
Motorola Mobility
Plantation, Florida, United States
(ewb001, edquarez-rivero, edh031)@motorola.com

Abstract

Project management and process improvement are a critical part of software development in an organization, especially for large scale and long-lived software. Metrics can be used as one of the means to evaluate this process. However, traditional methods of measuring effort, for example, focusing on editing time and lines of code produced, do not reflect the true cost to the organization. The organization pays developers for their time, regardless of whether they are writing code or performing other activities. This paper describes an experimental approach to track and measure effort in developer days, using log files from a corporate repository. The data is fine-grained, empirical, and non-invasively collected. Results have been tested on several projects in a large organization over a period of many years.

Keywords: Repository, effort, metrics, project management.

1. Introduction

Effort is a core metric in both project management and process improvement. Earned-Value Management, for example, compares planned and actual effort to gauge schedule progress and slippage. Lean Process Improvement compares total effort to value-add when measuring productivity. When comparing one project to another, total effort can be used as an indicator of their relative size.

Effort, itself, is a fairly simple concept. It is measured in person-hours or person-days. When the cost-per-hour for each person is known, cost-adjusted effort can also be computed. When the type of work is known, effort and cost can also be classified by type. Unfortunately, in software development, measuring effort empirically and per activity is not always easy. Numbers self-reported by developers are sometimes used. However, such numbers are unreliable and only account for part of the developer's time and salary. The problem of assigning cost is further complicated in organizations where staff members switch back and forth among activities that sometimes overlap. The problem of determining effort is captured in the following excerpt from a published interview with a manager concerning challenges in software development.

"The metrics are very hard to gather. One of the efforts under way right now is to figure out what it's costing us to deliver (the product) because before we had no clue. Time was being spent everywhere and it wasn't being put in granular buckets to know we spent this much on project management, this much on testing, this much on rework or whatever." [1]

Current methods of measuring effort in software development are based on surveys and self-reports. In one method, developers are required to fill out reports to account for their hours during the week. In another practice, when each task is completed, the developer must report how much time it took to complete the task. Both of these methods are subjective and fail to account for much of the time and effort. Neither method corresponds to the actual cost to the company.

To date, the published work on measuring effort with tool and repository data has measured only active coding time or numbers of lines changed. These methods do not correspond to the actual cost to the organization. Repositories offer an interesting alternative. Extracting metrics from the repository is non-invasive and low cost [2]. The data is both objective and fine grained.

In this paper we report on our experiments with repository data to produce empirical measurements of effort in units of developer days. Developers are paid for the full day, regardless of how they divide their time. Our method assigns the time in a developer's day to the tasks and projects on which the developer is active. The data used in this analysis was obtained from a large organization and covers several projects over a period of many years.

This paper offers three unique contributions. First, the data is from private industry. In this work we leverage features of the data that are characteristic of mature practices in private industry. Second, the measurement of effort corresponds to what the organization actually pays. Because developers are paid for entire days, regardless of what they are doing, our approach assumes that cost is based on days and uses it in assigning effort to projects, tasks, and artifacts. Third, our goal is to automate this process of measurement. Extracting data from the repository is often a manual process where the data is extracted, formatted, explored, and pruned. Here we classify the issues that must be addressed and provide an executable solution.

As an example of our type of result we can produce, Figure 1 shows a graph of effort on a project, per day, from inception to release. The X axis show the project day, starting from day 1. The Y axis shows the number of developers working on the project on that day. The lowest (red) line that rises in the middle and crosses the other (green) line, represents developers repairing bugs (rework). The middle (green) line that rises at the beginning, and drops in the middle, represents developers working on new requirements. The highest (blue) line shows the total number of developers, regardless of the kind of work being done. As explained in the paper, the numbers are adjusted to represent actual worker-day units of effort.

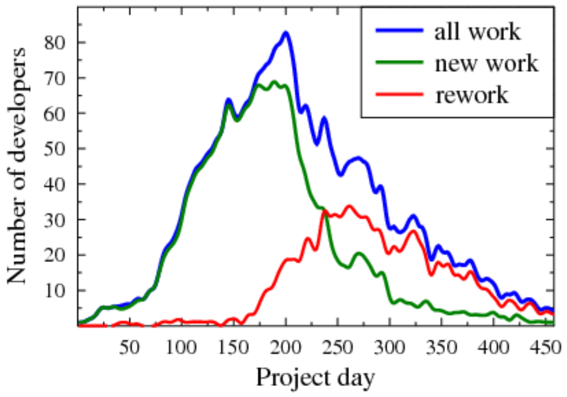

Figure 1. Number of active developers by project day.

The rest of this paper is organized as follows. Section 2 discusses related work and positions the work in this paper. Section 3 describes the important details of the research, including issues and resolutions. This section details the steps of transformation, from repository extraction to metrics display. Section 4 shows results for different projects while highlighting significant details. Section 5 concludes the paper with a discussion of broader implications and immediate next steps that are planned.

2. Related work

The existing research on measuring effort in software repositories differs from the work presented here in two important respects. The first difference is that the existing work is largely academic and based on data from student experiments and open source projects. In open source development, the work is often voluntary and performed by developers in their spare time. Requirements and contributed code appear at random intervals. There is no real relationship between effort and cost, and bug fix times are measured in weeks [3]. In our data, the relationship between effort and progress is direct and the average bug fix time is less than 3 days.

The second difference is that most of the existing literature on effort defines effort rather narrowly as "active time" spent editing source files. Both Johnson et al. [4] and Hochstein et al. [5] instrumented the development environment in order to identify times when the developer was actually typing. Times between active work periods were viewed as "interruptions" or "non-work intervals," and not counted. Measuring effort in this way makes sense if the goal is to determine which tool is the easiest to use. But when applied to industry, it fails to account for a large part of the developers' time, and the project's overall cost.

Perry et al. conducted several studies of how developers spend their time [6]. In their studies, they used self-reports and direct observation. Not surprisingly, they found that developers spend significant amounts of time in meetings, on the phone, reading and answering email, preparing reports, and creating documents. In an interesting comment, they noted that much of the observed communication did not involve technical matters, but instead concerned the process. Process costs of this type do not appear in measurements that include only "active time."

In this paper, our concern is not about how developers spend their time. Rather, our concern is simply to measure total developer time, regardless of how it is spent, and to count that time, as a cost attributed to specific projects, tasks, and artifacts. Only when the developer works on a different project, or does not work at all, is the time not counted. In private industry all of the developers' time should be accounted for.

While not specifically concerned with effort, there is other repository work that bears some relationship to the work presented here. Connecting data from bug tracking with data from configuration management is described in Fischer et al. [7]. They used a process that is very similar to the one described here – applying Perl scripts to the extracted files and storing the results in a database. They even faced a similar challenge of finding the task ID in configuration management records and again applied a similar solution – looking for the ID string embedded in other fields. In their data, the ID appears in the comment. In our data, the ID appears in the comment or the branch name. Our work differs in that we describe the process of identifying effort. In their case, the work produced release histories for artifacts. We also look at release histories, but not in the work described here.

Part of the work presented here concerns automating the process of collecting repository data and reducing it to produce the effort metrics. A number of papers describe projects to automate the collection of repository data. Bevan et al. for example, describe a system they call Kenyon [8]. In their work, the main focus is on capturing code changes. They mention capturing the developer ID (author), but only in a discussion of making such "metadata" consistent across different tools. Similarly, they are concerned with correlation between commits of different artifacts, but not between records in different kinds of tools.

3. Our research

The goal of our work is to extract metrics from the repository that can be used for both project management and process improvement. In private industry, managers want to know how much things cost. They want to know how much effort is spent on work and how much is spent on rework. At any given time, they want to know who is doing what, and how it compares to prior estimates. When improvements are made, they want measurements to show if things are getting better. All of this information requires tracking effort.

Our hypothesis is that it is possible to construct reliable measurements of effort, with daily and per-artifact granularity using data from the repository of a development organization in private industry. We make an important assumption about the data, that it reflects an organization with reasonably mature practices. Our goal is to construct such measurements automatically and on demand.

Effort is measured in developer days or hours. For a given day, the effort is the number of developers working on that day. To compute this effort, we leverage the fact that our data comes from private industry. In a business, developers are on a payroll. If they are active, they work for the business and generate repository events on a nearly daily basis. Project effort can be computed daily by counting the number of developers working on that project on that day. The challenge is to figure out which developers are active and on which tasks and projects they are working. In cases where developers are working on more than one task and/or project, or are switching between projects, the overlap must be detected and addressed.

As previously mentioned, our process assumes that the data is from a mature organization. Mature organizations have practices and enforce policies that assure order and avoid trouble. These practices affect the quantity, quality, and reliability of data in the repository. Specific practices and their implications are described below. The teams whose data we used were certified at the Software Engineering Institute's CMMI level 3 or higher. The work reported here uses data from historical records maintained by Rational ClearQuest and Rational ClearCase.

3.1 The Data

In a mature organization, all development work must have a reason and be associated with a documented work request. Work requests, or tasks, are maintained by a task tracker. A task tracker record includes the task, the project, the date submitted, the date resolved, and the type of task. The two main types of tasks are new requirements and defect repairs. A record in the task tracker may contain additional information such as assigned developers, the developers' reported hours (as indicated above), and the identity of the test that found the bug. For this work, we needed only the task's ID, its project, the task type (requirement or fix), and the resolved date.

In the Rational repository, all development work is done on separate branches. The main product branch contains only approved and tested code. When a developer starts work on a new task, a new branch for development is opened. Work continues on that branch until all changes are tested and complete. Only then are the changes merged with the product back onto the main branch. Thus we see the beginning of an activity in configuration management logs as the creation of a branch. The continuation of the activity appears as the checking in of versions of artifacts on that branch. Records from the configuration management log contain a date and time, a developer ID, the event type (e.g. "create branch"), the name of the branch, and the name of the artifact. There is also a comment field. For this work we need only the event date and type, the branch name, and the developer ID. Once we have the raw data, work proceeds in three phases: preprocessing, data reduction, and data extraction. As mentioned earlier, our data came from a ClearQuest task tracker and ClearCase history logs. Similar data would be found in Bugzilla and CVS or SVN. The vendor in our case (IBM) provides a tool for extracting the raw data, including some formatting options.

The projects in our study are substantial, with up to 100 developers, task counts in the thousands, and numbers of configuration management events approaching six figures. The exact size, in terms of lines changed or function points, is proprietary and cannot be published.

Our goal is not to create a benchmark for other corporations or entities, nor to indicate that the organization involved in our study is doing better or worse than others. Organizations have different people, different practices, and projects. Rather, the data is used to create an organization's own baseline and metrics to plan and track improvement. From one project to the next, within an organization, the people, practices, and projects are not so different. The contribution in this paper concerns the process of deriving the metrics, and their ability to shed light within the organization from which they are collected.

3.2 Preprocessing

Most of our analysis work is done using a database. In the initial preprocessing step we transform raw data, obtained from a software repository, into forms that can be loaded into database tables. The database dictates formats for date and time and puts limits on the size of various fields. The order of the fields is also fixed. Each table field must be given either a meaningful value or the database equivalent of a null value. Preprocessing "cleans up" data, formats it correctly, infers values that are not directly given, and excludes data that is not needed.

The two main challenges in preprocessing are: 1) extracting the data needed for correlating configuration management events with task tracking records, and 2) determining which repository events represent valid work. We discuss each problem in turn**.**

In mature organizations, it is common to have a policy that code cannot be changed without a corresponding task. Unfortunately, most configuration management tools do not support, let alone enforce, the connection to a task tracker task. However, this functionality can often be added. For many tools, example scripts can be found on the Web. In our case, the connection between a branch and its task was enforced only by a convention for naming branches. The name of the branch included information that identified the task. Preprocessing extracted this information, including variations and misspellings, and generated the needed field for task ID. In some cases, a branch is associated with more than one task. When this occurs, we simply duplicate the event for each task. The method of counting effort, described below, addresses multiple and overlapping tasks, so that double counting of the effort does not occur.

Not every event in the repository represents work. Some repository events reflect activity by tool administrators or automated scripts or daemons. In our metric, effort is activity by a developer working on a task. Each episode of work has a beginning, a period of continuation, and an end. In preprocessing, we add a tag that specifically identifies non-developer and non-development work. Several heuristics are required, including the type of event, fixed and stylized comments, specific branches, and the types of artifacts involved. We eliminate records from certain administrative acts and any activity associated with automatic processes.

One type of event not found on a development branch is still needed to complete the analysis. When a task is completed, the work on a development branch is merged back to the product branch. This special event marks the real completion of the task. The event appears as new artifact versions, by a select group of developers, on the main branch of the product. Unless it is included in the comment, these main branch records lack any reference to a task ID. The problem, and our solution, is very similar to one described in Fisher et al [7]. In the preprocessing step, we identify all version events on a main product branch and give them a special tag for later processing.

A Perl script was used to automate the preprocessing phase. Automation makes it possible to make adjustments and quickly repeat the entire phase. The preprocessing script generates a separate SQL script to load the data into database tables.

3.3 Reduction

In the reduction phase, we combine the task tracker and configuration management data into a single work event table. A work event is one developer working on one branch for one task over a contiguous period of time. A record in the work event table includes the developer, the branch name, the first and last dates of valid change events, the task, and the type of work (requirement or repair). The table is specific to a single project.

Using branches to represent work proves very convenient. On a branch, work happens between the date when the first artifact is checked in and the last artifact change occurs. But there are issues to be resolved. A developer may be working on several branches at the same time. More than one developer may be using the same branch. Finally, work may start on a branch, be interrupted, and resume at a later date. We address each of these issues in turn.

We often find instances where a developer has several branches active at the same time. While performing work, developers may open additional branches. Branching can be used to separate instrumented from non-instrumented code, to try alternatives, or to test changes against different product variants. In the extraction phase, we count a developer as being active if they have work that is active on any branch. Work is allocated by developer, not by branch. Thus, there is no danger of double counting effort when the same developer is active on more than one branch.

A special case of developers with multiple branches occurs when a developer is active on branches for more than one project at the same time. In this case, we have to decide if the developer is multi-tasking, or if one branch was interrupted to work on the other project. The determination is made depending on whether events on the two projects interleave, or all of the work on one project fits entirely between two stretches of work on the other project. In the former case, where there is multitasking in active periods that include the same day, we divide the time evenly among the tasks involved. In the latter case where there is an interruption causing the developer to work on a different project, the surrounding stretches are split into separate work events. Because we have data on all projects (it is in the same repository), we can do the necessary comparison, even with tasks in different projects.

Two developers may work on the same task, or, though rare, even the same branch. For branch begin and end dates, we group the events by both developer and branch. Thus each developer's work is treated separately with their own begin and end dates**.** These dates are then used to decide, on a given date, which developers are working in some capacity on the current project.

When we compare dates reported in the task tracker with actual dates found in the configuration management event log, we face a new issue. Work occurs on branches after the date when the corresponding task was reported as resolved. Three possible explanations are considered: 1) the task tracker data was incorrect, 2) the task tracker data was correct, but final adjustments were needed before merging with the main branch, or 3) work that was attributed to the task was not being done for the current project.

In the first case, task tracker reports are subjective and might not be entirely accurate. Work actually continues for 1 or 2 additional days after the report of resolution. In the second case, the task work has passed inspection and acceptance testing. But before the work can be merged back into the main branch, last minute adjustments are needed to resolve conflicts. The work typically appears within 10 days of the reported completion, does not involve a new branch, and is immediately before the merge. In either case, the work is counted towards project effort.

In the third case, the existing task ID is used for work that ultimately becomes part of a different variant, or future product release. Typical scenarios could be that the original fix proved valuable and was quickly applied to other product variants without opening a new task. In an alternate explanation, a better solution was found and tested, but only for use in a later release. In both of these cases, the work has its own sub-branch, and is begun later than the reported resolved date. The work being done is also never merged with the current project's main branch. In these cases, the effort is not counted towards the current project**.**

The reduction phase is performed by an SQL script and output to a single work event table containing all the data for one project. The work event table includes the developer ID, the branch name, the first and last dates of the developer's work on that branch, the task ID, and the type of work (requirement or repair).

3.4 Extraction

In the last step, we count how many developers are actively working for the project on each project day. The output is an effort table with one record for each project day. An effort record has fields for calendar day, number of active workers, number of workers active on requirements, and number of workers active on repairs. To perform the adjustments for overlapping work, we count workers involved in both development and repair tasks, workers involved in development and non-project tasks, workers involved in repairs and non-project tasks, and workers involved in all three types of work (development, repair, and outside the project). The last case is very rare. These last counts are used to adjust the data, dividing the effort so that the totals match the actual number of developers.

At this point, the major challenge is deciding what to count as a project day. If we count every calendar day, we would also count weekends and holidays when no work is actually being done. It seemed better to count just the traditional work days – Monday through Friday, minus holidays. But looking in the repository, some weekends had significant activity. A project could show less effort by having work done on weekends, and thus not counted as effort. In the end we included weekend days which, after adjusting for work in different time zones, showed evidence of significant work. We excluded holidays when there was relatively little work.

A developer is counted as working if they have a work event record (as described in the previous section) where the current day falls between its first and last day of work. As explained earlier, a developer may have multiple branches within a task, and may also be working on more than one task, in which case only part of the day is counted under each activity.

3.5 Display

After extraction, the data is ready for display. The daily records of effort are output to a Comma-Separated-Value (CSV) text document, which is then loaded into a spreadsheet. The data is then displayed. A sample result is shown in Figure 1.

Because we adjust for overlapping effort on different projects, tasks, and task types, the line for all developers is in fact the sum of the other two lines (development and repair). Since a few developers may be working on both requirements and repairs at the same time, without adjusting the allocated effort, a simple sum would have counted them twice. Similarly, the effort of workers active in more than one project might otherwise be counted twice between the projects.

Days in this graph are project days, not calendar days. In general, weekends and holidays are excluded. But busy "non-work" days were included, as explained above.

To draw the graph in Figure 1, we smoothed the data using a 7 day sliding window average. Without the smoothing, daily fluctuations obscure the longer trends. The 7 days was chosen as the window size to also smooth over variations due to the day of the week. Figure 2 shows the same data, prior to being smoothed.

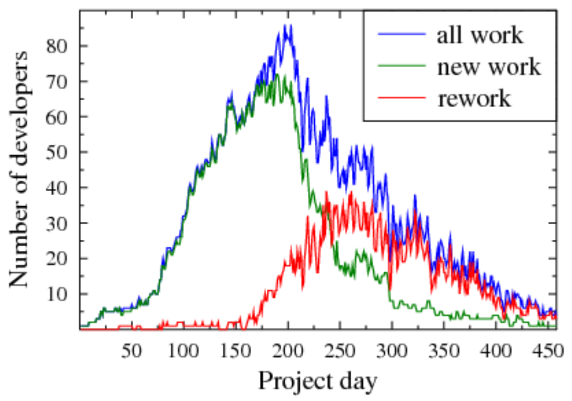

Figure 2. The same data as Figure 1 without smoothing.

The project's total developer effort, measured in person days, is the area under the curve. This area is easily computed by summing the daily counts over the life of the project. For the project in Figures 1 and 2, total developer effort was 14,600 developer days. For new work and rework, the numbers were 9,700 and 4,900 respectively. Rework accounted for 33% of total effort. At the time of this project, 33% was slightly better than average for similar projects within the organization.

4. Results

In the previous section we described how the data is collected and transformed to produce effort metrics. Here we discuss our experience and some uses of the data.

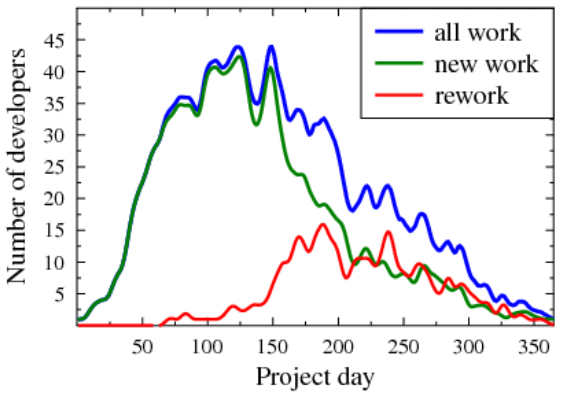

Figure 3. Effort graphs for a recent project with better practices.

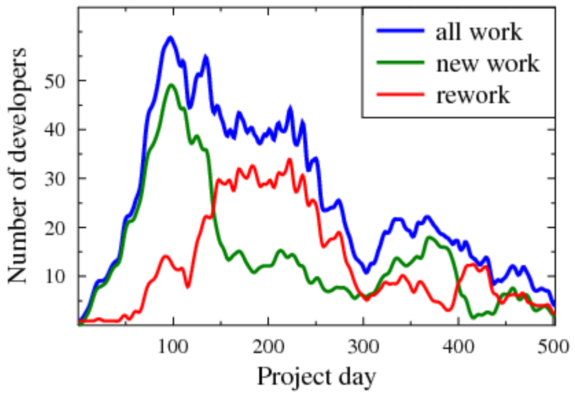

Figure 4. A troubled project with major effort for rework.

Figures 3 and 4 show effort curves for two other projects. The project in Figure 3 is a more recent project. Total effort was 7,700 developer days. In this case, rework effort was only 22%. We have not investigated all of the differences to account for this improvement. But the organization has improved its use of unit testing. Another difference, visible in the graphs is the steeper ramp up at the beginning and ramp down at the end, showing better time utilization, partly due to better coordination among projects.

The project in Figure 4 was an older project that experienced significant difficulties. The effort ramped up quickly in the beginning, but soon ran into trouble. Developers were pulled from requirements early to focus on rework. Another interesting feature, visible in the data, is a design change around day 300, resulting in a second hump on the line for new work. Total effort was 15,650 developer days. But rework accounted for 48% of total effort.

Views like those presented here are not currently available to managers in our organization. Current measurements of effort are based on reports collected from developers. We had access to these reports for some of the projects in the study. Overall, the reports varied widely. Many showed little relationship to the evidence in the repository. More significantly, as we reported in an earlier paper [9], they only account for 20% of the developers' time. When we compared developer days with hours reported for thousands of tasks, short and long, the 20% figure was surprisingly consistent. We don't know if it is a coincidence, but 20% matches the industry average value-add efficiency for development engineers reported by Kennedy [10]. Developers are paid for days at work, not just self-reported hours of value-adding activity. Thus the method of measurement presented here is a more realistic indicator of cost.

Starting with the views shown, and based on the same methods of data reduction and analysis, our approach presents many opportunities for further drill-down analysis. It is possible, on any day, to determine who is working on what project, which task, and which type of task. That data is actually present, and in that exact form, in the reduction step. Managers we spoke to said they often wanted that information, but cannot get a reliable report. From the same data, managers could also determine which tasks, are started but not yet complete, have been running for the longest time and are consuming the most effort.

By changing the grouping criteria, again in the reduction step, the approach could be used to divide the effort by product subsystem or component, or types of artifact. Hindle, Godfrey, and Holt [11] describe a method of partitioning among document update, test preparation, and other activities. Their method could be applied with the data here.

5. Conclusion and future work

We had produced a graph, much like the one in Figure 1, during an earlier phase of our research [9,12]. But the data from that one project was cleaned manually, and the metrics for effort was extracted only after many intermediate steps of data reduction and transformation. In the work presented here, we wanted to demonstrate that the metrics for effort are generally available, and that the process of extracting them from common repository data could be automated. We are now able to produce these graphs for any project in our repository, at will.

When publishing details of our work, we frequently encounter the question of whether the numbers we produce are comparable across organizations. These metrics depend on specific reporting practices, the way in which work is organized, and the individuals involved. But, in process improvement, the goal is to do better than we ourselves have done in the past. The processes, practices, and people within an organization stay pretty much the same. Thus the metrics are highly reliable for internal and longitudinal measurements of process improvement. While specific details may vary, what is generalizable across the industry, is the general method of extracting data and presenting it in a view. Moreover, practices of using branches for development work and associating changes with tasks, are themselves best practices and not unique to the organization that was studied. We believe that our method of allocating and measuring effort can be applied beyond the organization in this study.

In many fields, project managers use the Earned Value (EV) method of tracking progress. Earned Value counts the effort of a task as its value, and tracks how much of the expected effort has been achieved to date. It is difficult to apply the Earned Value method of management when your only metrics are hours spent, lines written, and defects not yet fixed. In the future, we hope to show that we can produce reliable metrics to support this type of management.

Our work began with efforts to use repository data for objective, and actionable measurements of the process using the types of data available in private industry [13]. We are succeeding in that effort. More recent work has focused on methods of visualization [14]. At this time, our effort is shifting to put these kinds of tools directly in the hands of the organization's decision makers. The work presented here is a step closer to that goal.

6. References

[1] A.R. Hevner, R.W. Collins, and M.J. Garfield, "Product and Project Challenges in Electronic Commerce Software Development", *SIGMIS Database* 33(4), 2002, pp. 10-22. DOI=10.1145/590806.590810

[2] S. Huang, S.R. Tilley, M. VanHilst, and D. Distante, "Adoption-Centric Software Maintenance Process Improvement via Information Integration", *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, 2005, pp. 25-34.

[3] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How Long Will It Take to Fix This Bug?", *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR)*. IEEE Computer Society, 2007.

[4] M. Johnson, K. Hongbing, M. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita, "Improving Software Development Management Through Software Project Telemetry", *IEEE Software*, 22(4), 2005, pp. 76-85.

[5] L. Hochstein, V.R. Basili, M.V. Zelkowitz, J.K. Hollingsworth, and J. Carver, "Combining Self-reported and Automatic Data to Improve Programming Effort Measurement", *Proceedings of the 10th European Software Engineering conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* ACM, 2005, 356-365. DOI=10.1145/1081706.1081762

[6] D.E. Perry, N.A. Staudenmayer, and L.G. Votta, "Understanding and Improving Time Usage in Software Development", in Volume 5 of *Trends in Software: Software Process*, John Wiley & Sons, 1995.

[7] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems", *Proceedings of International Conference on Software Maintenance (ICSM)*, IEEE Computer Society Press, 2003, 23–32.

[8] J. Bevan, E.J. Whitehead, S. Kim, and M. Godfrey, "Facilitating Software Evolution Research with Kenyon", *Proceedings of the 10th European Software Engineering conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, ACM, 2005, pp. 177–186.

[9] M. VanHilst, S. Huang, and H. Lindsay, "Process Analysis of a Waterfall Project Using Repository Data", *International Journal of Computers and Applications*. ACTA Press, 2011.

[10] Kennedy, M.N. *Product Development for the Lean Enterprise: Why Toyota's System Is Four Times More Productive and How You Can Implement It,* Oaklea Press, 2003.

[11] A. Hindle, M.W. Godfrey, and R.C. Holt, "Release Pattern Discovery via Partitioning: Methodology and Case Study", *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR)*. IEEE Computer Society, Washington, DC, USA., 2007.

[12] M. VanHilst and S. Huang, "Mining Objective Process Metrics from Repository Data", *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering* (SEKE), 2009, pp. 514-519.

[13] M. VanHilst, P.K. Garg, and C. Lo, "Repository Mining and Six Sigma for Process Improvement", *SIGSOFT Software. Engineering Notes* 30(4), 2005, pp. 1-4.

[14] S. Huang, and C. Lo, "Analyzing Configuration Management Repository Data for Software Process Improvement", *Proceedings of the 19th IEEE International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2007.