Lessons from Software Work Effort Metrics[1]

Karl E. Wiegers

Process Impact
716-377-5110
www.processimpact.com

How many of these questions about your software development organization can you answer with confidence?

1. How much could your development productivity be improved if you did not have to spend any time on maintenance?
2. How much did it cost you last year to adapt your software portfolio to work with new versions of compilers, database managers, or operating systems?
3. Which of the applications you support demands the most user support time?
4. What fraction of your time is really spent on testing?
5. Do you believe your developers spend enough effort on design activities?
6. Has your software development process matured over the past few years?
7. Are the efforts you devote to improving software quality reducing the time you must spend correcting bugs?
8. How accurately can you estimate future projects?
9. How many different projects did each of your developers work on last year?
10. What is the average number of effective project hours per week for your staff?

Many software engineering organizations will find it difficult to provide quantitative answers to these questions. Yet this level of understanding of the way an organization operates is necessary to identify key improvement opportunities.

In his article "Measuring 'Rigor' and Putting Measurement into Action" (*American Programmer*, September 1991), Howard Rubin defined ten basic categories of metrics that can be used to measure software development organizations, projects, and processes. Two of his categories are work profile and work distribution metrics, both of which pertain to the ways in which software work is distributed among different development and maintenance activities.

Any kind of productivity metric also requires that one knows how much work effort went into creating a product. Historical measurements of productivity are needed to prepare accurate estimates for future projects, a task that Capers Jones identified as one of the five critical software project management tasks in his article titled "Applied Software Measurement: The Software Industry Starts to Mature," *American Programmer*, June 1991.

This article describes the experience of a small software group that supports an industrial research organization. (Our general approach to software development was described previously in my article "Implementing Software Engineering in a Small Software Group," *Computer Language*, June 1993.) Since 1990, we have measured the time we spend in different work phases on all new development and maintenance projects.

---

[1] This article was originally published in *Software Development* magazine October, 1994. It is reprinted (with modifications) with permission from *Software Development* magazine.

By tracking trends in the distribution of these activities, we have improved our understanding of how we develop software. We have used the data to set quantitative improvement goals, identify leveraging opportunities that can increase productivity, and develop heuristics to assist with estimating new projects. Our database now represents over 50,000 staff hours of work.

Collecting work profile and work distribution metrics has become a part of our software engineering culture. It is not as difficult as you might think, and the lessons you can learn from the data constitute a valuable tool in your continuing quest for improved software quality and productivity.

Work Effort Classification

We classify work effort into one of six development and four maintenance phases. The development phases are: preliminaries and project planning, specification, design, implementation, testing, and writing documentation. The maintenance categories are: adaptive, corrective (fixing bugs), perfective (adding enhancements), and user support. One other category is used for capturing time spent on routine execution of certain programs, such as generating monthly management reports from an information system.

Tables 1 and 2 describe some of the activities that are recorded in each of these ten work effort phases. Some tasks cannot be neatly compartmentalized, so conventions were established to ensure that all team members report similar activities in a consistent way. For example, work with a user interface prototype could logically be attributed to specification (if the purpose of the prototype is to elicit requirements), or to design (if the purpose is to explore possible solutions), or to implementation (if the prototype will evolve into the product).

The use of these “phases” does not necessarily imply a waterfall software development life cycle. Most software work can be classified into one of these ten domains, regardless of the life cycle model being used. The main difference between one model and another is the sequence in which the activities are carried out. For example, in an incremental delivery model, you do a little specification, design, implementation, and testing for the first increment, followed by another such sequence for the next increment.

Data Collection Philosophy and Process

Each team member is expected to keep daily records of the time he or she spends on each project activity, to half-hour resolution. For accuracy, work activities must be recorded in real time, rather than trying to reconstruct them from memory at the end of the week or month. Time spent working at home or outside normal working hours is captured as well. The objective is not to add up to 40 hours per week, but to understand how much work time is actually spent on each project.

Only time spent on software project activities is recorded. Time spent in meetings not associated with a project, or any of the thousand other ways that time evaporates, is not captured. This is a major difference between a true work effort measurement program and an accounting process disguised as a metrics program. While both can be important, we feel that our approach provides a more accurate understanding of the actual work that went into a project.

Applications that have not yet been officially released have only development time recorded. Time spent finding or correcting bugs, or adding new features, is counted as testing or implementation until the application is installed into production. Similarly, maintenance work on

a released system is not subdivided into development phases, unless a different project name is being used for a major enhancement effort.

For systems that are released incrementally, we classify time spent working on a released increment as maintenance and that spent on an unreleased increment as development. This released/unreleased demarcation avoids the recursive complexity of slicing every enhancement into tiny chunks of specification, design, implementation, and testing.

The metrics data is stored in a simple Oracle relational database on a mainframe. Team members enter their own data using a simple on-line form. To reduce the number of entries in the database, work summaries are entered weekly, with all of the hours spent that week on the same phase of the same project entered as a single record.

The process of recording and reporting work activities is not burdensome. The group members report that they spend an average of only about ten minutes per week on these steps.

Some concerns have been raised about the temptation for team members to report inaccurate data in an attempt to look better to management. However, the metrics recording seems to be a self-controlling process. If someone reports fewer hours of work than he really spent, to increase his apparent productivity, the question arises as to where the remaining hours in the week went. Conversely, if more hours are reported than actually were worked, one's productivity appears lower than it should. In practice, the integrity of our team members has superseded any inclinations to distort the data.

Reporting Options

The work effort metrics we collect belong to the team members, not to the supervisor or the software process zealot in the group. The team members should not view the metrics effort as a write-only database, in which data goes in, never to be seen again. A simple reporting tool is available that lets each group member generate a variety of summary reports. Each member can also browse through his own raw data. To protect the privacy of individuals, team members cannot directly view data belonging to anyone else. However, anyone can generate summary reports by project, or for the entire group.

There are three main reporting options:

1. Generate a summary distribution of work effort by phase over all projects during a specified time frame, for an individual or for the group as a whole.

2. Generate a summary distribution of work effort by phase over the duration of a particular project, for an individual or for the group as a whole.

3. List the hours and percent of total time spent on each project during a given time period, for an individual or for the group as a whole (useful for identifying the myriad small tasks that siphon time from one's principal assignment).

In addition, the team leader can perform ad hoc queries and generate additional reports, such as showing the hours reported by week for each team member. This is most helpful for seeing when data has not been entered into the database.

We generate summaries from the first reporting option on a quarterly basis. This information is added to the historical quarterly summaries to see how the distributions are evolving. These reports are shared with the entire team and with management. In this way, we

all have a better understanding of how our group is building software today, as well as identifying problem areas and having an opportunity to think about what we should be doing differently.

Trends and Application

Collecting data for a few people over a short period of time will provide a snapshot of the group's work effort distribution that may or may not be representative of long-term trends. We found that about two years of data was required to have the cumulative distribution approach some steady state. Of course, we looked at the earlier data for clues as to what our major improvement opportunities might be.

A complication of any kind of software process measurement is that the target is moving (at least, it should be, if you are seriously pursuing continual process improvement). Further, the composition and activities of the group changed somewhat as we merged with other groups and absorbed their historical baggage of older, high-maintenance systems. These factors must be taken into account when interpreting long-term trends.

We chose to use the metrics data as a driving force for change. Consequently, the distribution of work for our group for all of 1993 looks quite different from that for 1990, and suggests that our development process has matured over the years.

Figure 1 shows how the distribution of our development work has changed in the four years since we began collecting data. The development data for each year has been normalized to 100%, as if no maintenance work had been performed. These are the notable trends:

- Design and testing efforts have steadily increased over the years. This is not an accident. After collecting data for two years, the group felt that our products would benefit from increasing the relative effort in these two areas. We set, and met, a goal for 1992 of increasing both design and testing effort by 50% over their cumulative 1990-91 values. The continued growth in 1993 indicates that the behavioral changes we adopted to achieve our initial goals have become instilled in our routine activities. These trends provide a quantitative depiction of a software process improvement.

- The fraction of time devoted to implementation has trended downward, as more effort has been focused on the upstream activities of specification and design. A corresponding decrease in defect correction (Figure 2) suggests that we are building better software with the help of better software requirements specifications (SRS).

- Specification and implementation reveal complementary trends, with one being high when the other is low. In a small group, such relationships are to be expected. The work distributions of a large organization working on many projects will be more nearly statistical, and the influence of individual projects will be less significant.

- An initial downward trend in preliminaries and project planning was reversed in 1993, when we began to concentrate more on formal project management. Again, process evolution will clearly affect the trends in your work effort distributions.

Figure 2 illustrates corresponding trends in the four maintenance activities over the years (again, normalized to 100% for each year). The most notable change is the large reduction of debugging effort in released programs. The fraction of our total software work devoted to defect correction has dropped from 13.5% in the first quarter of 1990 to a steady state of less than 2%.

This quality improvement is attributable to several factors, including: extensive end-user involvement to create a better SRS; more design and testing effort during development; implementing design and code inspections; and more formalized software quality assurance practices. Besides giving us confidence that we are creating high-quality products, this improvement frees up time to be spent on new development.

Figure 2 shows that our greatest maintenance challenge is adaptive maintenance, which continues to consume nearly 8% of our total project work time. As adaptive maintenance usually is stimulated by factors outside our control, it is difficult to reduce. An emphasis on portability and reusability will help, as will using mainstream development languages, compilers, and databases. But no one is immune from version N+1 of the commercial software products they use, nor from the need to migrate applications from one platform to another. One relational database vendor has been a particularly annoying contributor to our adaptive maintenance burden.

Table 3 summarizes the fraction of total work time devoted to development versus maintenance each year. The original team merged with another group that supported a low quality legacy system at the beginning of 1992, which explains the big jump in maintenance effort that year. The trend toward reduced maintenance resumed in 1993, however.

The software literature cites a wide range of numbers representing the fraction of total effort (time and money) spent by IS organizations on maintenance, ranging up to 80%. In Capers Jones' definitive book *Applied Software Measurement*, he reports that U.S. software work is split into about 41% new development, 45% enhancement projects, and 14% corrective maintenance The definition of "enhancement" is quite subjective.

Clearly, our group has a much lower maintenance burden than these figures represent, more in the range of 18% for all types of maintenance (at least by our definitions). If we made project estimates or decisions about productivity-enhancing products to buy based only on industry averages, we would be way off base. If you want to know how large your maintenance burden truly is, you'll have to measure it yourself. The data you collect also will help you identify problem areas on which to concentrate to most effectively reduce your future maintenance load.

Metrics-Based Project Estimation

Estimating the duration and cost of new software projects is a major challenge. Many available estimating tools rely on models such as COCOMO, which was described in Barry Boehm's classic 1981 book *Software Engineering Economics*. COCOMO is based on a sample of 63 projects done in the 1960s and 70s. How accurately will such models predict *your* organization's delivery capability?

Estimating work effort requires that you know the amount of software to be written, which is itself very difficult to estimate. COCOMO is based on the projected number of lines of code in the final product, which is not easy to predict early in the project. Metrics relating to the amount of functionality in a system are more meaningful early predictors of software size. Such functional metrics include function points, feature points, and the "Bang" metric introduced by Tom DeMarco in his book *Controlling Software Projects*. Functional metrics can be estimated from user requirements, screen mockups, or analysis models, which are available early in the project cycle.

The most reliable estimates are based on measurements of your group's historical performance on similar projects, adjusted for individual staff capabilities, new technology learning curves, and so on. Pages 168-169 of DeMarco's book list many factors to be used in adjusting predictions made from models.

We approached metrics-based estimating by exploring the relationship between the number of specifications in the SRS and the number of post-specification phase work hours needed to deliver the system (i.e., all time counted as design, implementation, testing, and documentation). We had been writing spec documents in a similar fashion for several years. The specs are the first thing done in the development process, and preliminary versions can be generated early in the project's life cycle.

SRS documents from eight projects were examined for consistency of style. A simple plus-or-minus 50% factor was applied to the raw count of functional requirements for documents that were written in especially dense or sparse styles. These activities represent a range of tiny to medium sized projects, and include utilities, reusable components, real-time lab automation systems, and information systems.

Figure 3 shows the correlation between the number of post-specification development hours and the adjusted number of functional requirements for these eight projects (circles). Using a linear least-squares fit and forcing the line through the origin (no requirements take no time) gives a slope of 5.6 hours/requirement, with a high correlation coefficient of 0.986.

Clearly, some requirements can be implemented in minutes and others may take days, but Figure 3 indicates that for projects such as the ones in this sample, an average requirement can be fully implemented, tested, and documented in five to six hours. This correlation includes systems developed in both waterfall and incremental fashion.

There is always the concern that a relationship this linear is a fluke. New project data points have been added since the original study was done (squares in Figure 3). These new points fell very close to the original line, without any adjustments. Thus, we conclude that in our little world, this correlation is the best available predictor of the effort required to deliver a system similar to those we have built in the past.

Not all projects estimated by this relationship have adhered to the initial line. For example, a reusable graphics component (triangle in Figure 3) required nearly twice the estimated work time, largely because much more extensive testing was performed than on older projects. This again indicates that, as your software process evolves, the oldest information in your historical database will become a less relevant predictor of future results.

Adjustments also must be made in the initial prediction to account for unusual characteristics of a particular system. For example, portability was a critical attribute of this graphics program, so extra design effort had to be devoted to achieving this objective. Similarly, reusable components often are tested more rigorously than are end-user applications, so extra test time should be factored in. Of course, we didn't realize all this when we did the original estimate for the graphics program, so our estimate was, as usual, too low.

The next step was to translate an estimate of work hours into calendar time. The metrics database again was valuable. It showed that each team member reports an average of 31 hours of effective project time each week (up from 26 hours in 1990). Consequently, the number of projected hours of work should be divided by 31, not 40, to estimate the weeks of labor required per staff member. Further adjustments for absences, work on other projects, and so on must, of course, be applied to the resulting number of weeks. The resulting model now forms the starting point for our new project work estimates.

Conclusion

This article shows that the collection and analysis of software work effort metrics provides multiple benefits to even a small software group. Many of the team members derive value from the summary data for project planning, estimation of tasks, and identifying improvement opportunities, such as activities that ought to have more time devoted to them. The data provides a quantitative understanding of the group's development process, as well as a way to monitor evolution of the process over time. It has been enlightening to many team members to compare where they *think* they spend their time with where they *actually* spend their time.

Tom DeMarco claims that you can't control what you can't measure. If you want to better control the way you build software in the future, find out how you are building it today.

Acknowledgment

The author wishes to thank Michael Terrillion, who established the original correlation between the number of functional requirements and the hours required for development.

Table 1. Development Phases and Activities.

Preliminaries and Project Planning	• Time spent prior to gathering requirements on activities such as defining the scope of a project, establishing the team and project champions, etc. • Time spent on planning and tracking activities throughout the project's life. • Training specifically required for the project. • Evaluating and selecting tools, computers, operating systems, and so on for a specific project.
Specification	• Working with customers to define the functions that must be contained in the system. • Writing and inspecting the software requirements specification. • Doing any necessary rework as a result of the inspections. • Creating and evaluating prototypes intended to more completely specify the requirements of the system.
Design	• Creating and inspecting high-level design models, such as data flow diagrams, entity relationship diagrams, and dialog maps. • Doing any necessary rework as a result of the inspections. • Designing user interface screens, algorithms, data structures, file formats, database schemas, and classes. • Creating and evaluating prototypes intended to determine whether the proposed system design is correct. • Defining program architectures. • Program design, including writing minispecs or PDL. • Evaluating, by whatever means, alternative solutions to any technical problem or user requirement.
Implementation	• Coding a program. • Inspecting the source code. • Doing any necessary rework as a result of the inspections. • Writing unit-level, internal module documentation. • Creating database tables, user interface screens, data files, or other artifacts that are part of the final system. • Correcting defects found during testing, but prior to delivery of the system.
Testing	• Writing and executing unit tests in a disciplined and systematic way. • Informal, ad hoc testing of the program. • Writing and inspecting software quality assurance plans and test plans. • Doing any necessary rework as a result of the inspections. • Writing and executing user interface or other system tests. • Designing and implementing automated test drivers and associated test data files. • Engaging in systematic alpha or beta testing. • Recording the results of formal test executions.

Writing Documentation	• Writing user aids (on-line help, user guides, reference manuals, tutorials, training materials). • Writing system documentation (i.e., external to the programs themselves). • Writing internal reports or giving presentations on the project.

Table 2. Maintenance Phases and Activities.

Adaptive Maintenance	• Making changes in software to cope with a new computing environment, such as a different operating system (as when porting a program), a new computer, a new version of a compiler, or changes made in an external system to which your system interfaces.
Defect Correction	• Making a feature in an existing software system work correctly, when it does not work correctly at present. The feature was always there, it just doesn't do what it should.
Enhancement	• Adding a new capability to a program, or modifying an existing capability to satisfy a user request for improvement.
User Support	• Answering questions about how to use a program. • Looking at a user's data to explain what is happening. • Explaining algorithms or assumptions in the program. • Helping a user move data from one program into another.

Table 3. Distribution of Work Between Maintenance and Development.

Year	Development %	Maintenance %
1990	82.5	17.5
1991	91.4	8.6
1992	71.2	29.8
1993	84.7	15.3

Figure 1. New Development Work Effort Distribution, By Year.

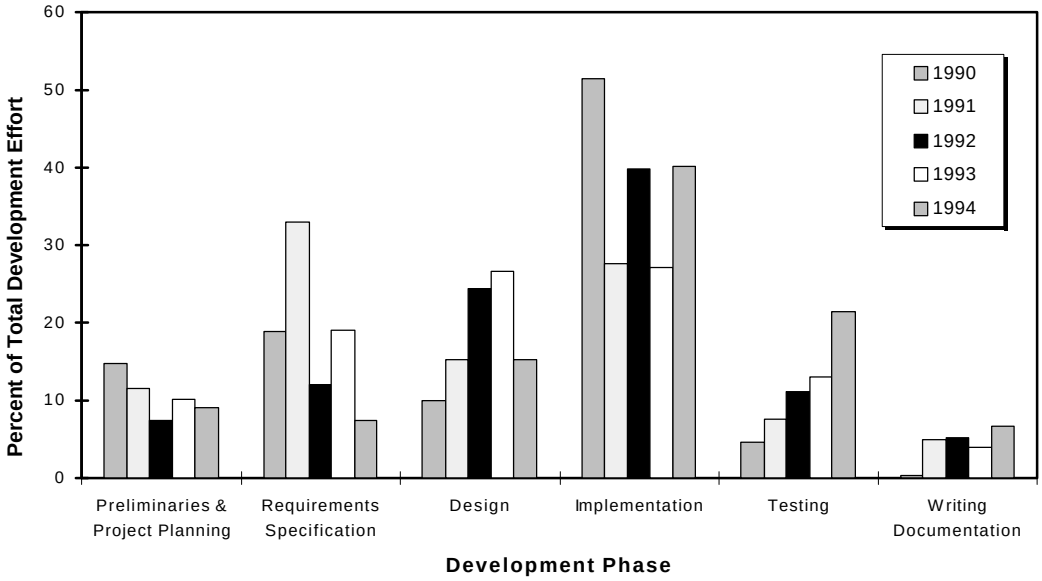

Figure 2. Maintenance Work Effort Distribution, By Year.

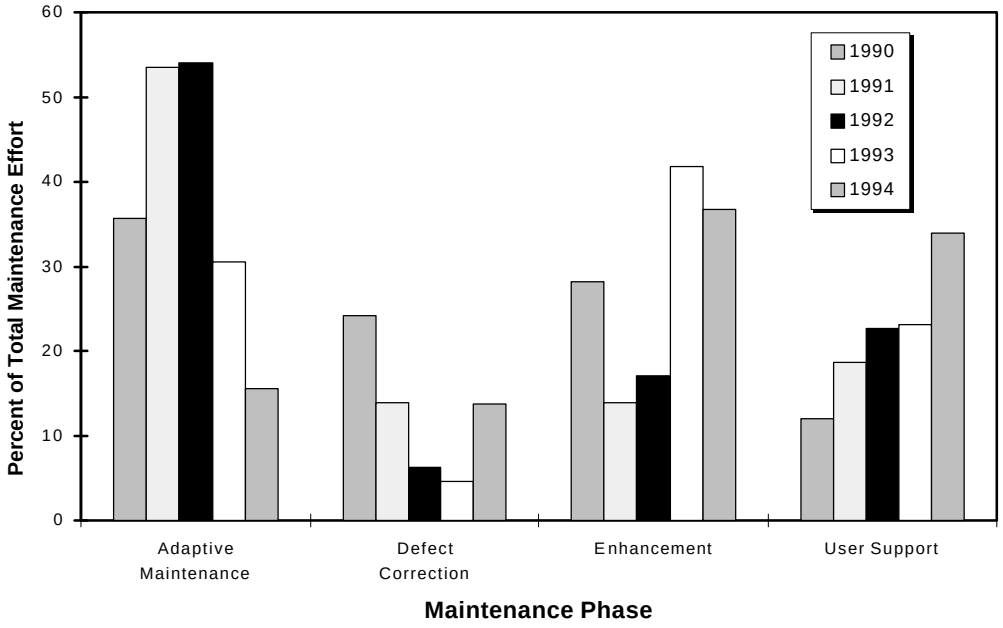
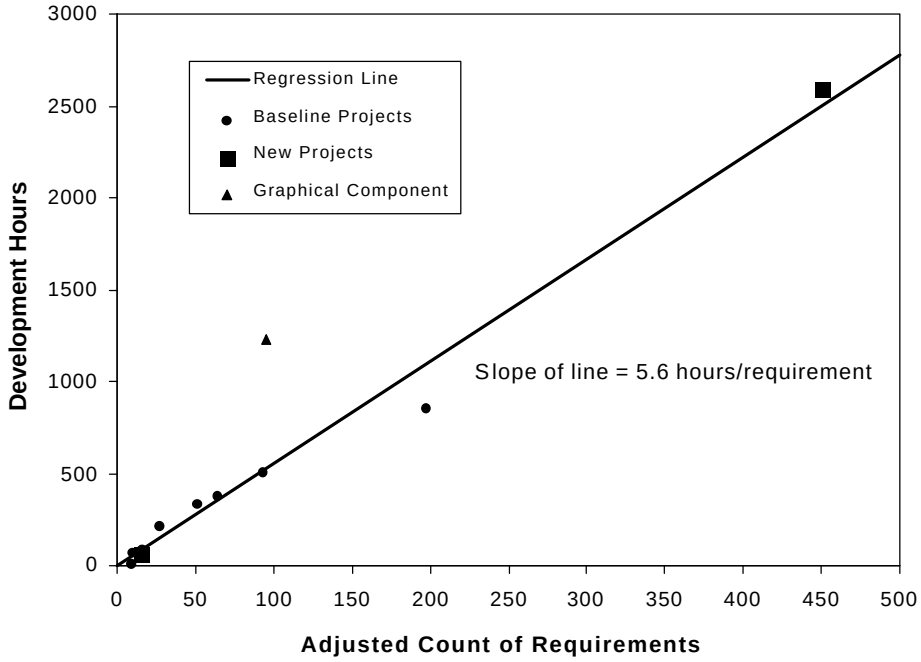
Figure 3. Development Time vs. Number of Specifications