'By Donald Anseimo and Henry Ledgard

Effective IT curricula balances tradition with innovation. One way to enhance that balance is to examine the common threads in the various knowledge areas.

MEASURING PRODUCTIVITY IN THE SOFTWARE INDUSTRY

"When you can measure what you are speaking about, ... you know something about it; but when you cannot measure it, ... your knowledge is of a meager and unsatisfactory kind... —Lord Kelvin

//... rapid growth in the power of hardware has ... permitted sloppy, unprofessional programming to become the virtual standard ... . Hardware has allowed the software profession to ... remain in an irresponsible adolescence in which unstable products with hundreds of thousands of bugs are shipped and sold en masse."
—Larry Constantine

Software is arguably the world s most important industry, according to Grady Booch. Software development environments used to build and support software are the major factor in software productivity. Yet, unlike hardware, there are no accepted measures that afford benchmark comparisons.

Gross measures presented in the literature indicate that software productivity has been dropping more rapidly than any other industry. The semiconductor industry had the most productivity growth (86) from 1990 to 1995. In that same period, productivity for the software industry decreased by 10, indeed, the worst decline of all industries surveyed.

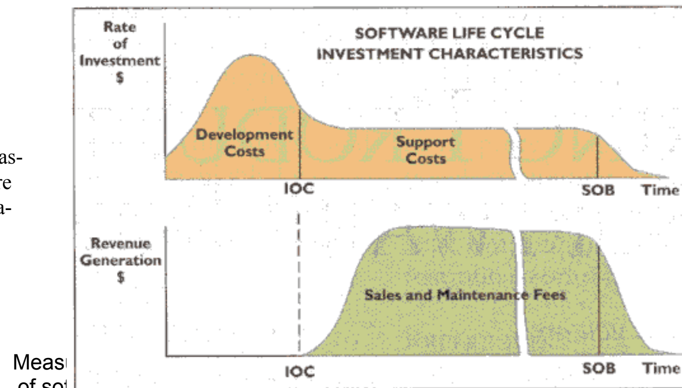
We have heard arguments that denounce these numbers, particularly regarding attempts for more sophisticated applications. This factor alone might account for negating a portion of the gross measure of productivity. On the other hand, one can argue the amount of memory we now use and the speed of hardware compensate for this. However, none of these arguments have a scientific basis. And that is the point of this article.

Borrowing from Tom DeMarcos Controlling Software Projects, "You can't control what you can't measure." Before we can expect to improve productivity, we must measure it. A framework is offered that we believe is essential for making improvements in software productivity. We start by addressing issues concerning productivity of software development environments.

We take for granted our ability to compare computer hardware productivity using benchmarks and purchase hardware based upon them. There are no acceptable productivity benchmarks for a software environment. Comparisons are generally based upon literature advocating a given method. Invariably they lack scientific data to support the claims.

Ability to deal with complexity. Software complexity grows rapidly when dealing with interactive user inputs, complex databases, dynamic graphics, networks, and so on. When functionality grows and software becomes more complex, development and support tools are put under the stress of a production environment. The more facilities contained in that environment to ease the development or these functions, the higher the productivity.

Scalability. The increasing complexity of software products stresses the scalability or the development environment in different directions. Various features of this environment can help or hinder the growth of an evolving product. What may appear unnecessary in solving a classroom problem may be critical in controlling the evolution of a large system in production.



Meas
of so
and support.

Approaches to producing complex systems not evolved in a production environment typically don't scale well. Intel's approach to chip design and fabrication is an example of the evolution of a good production environment. We question the ability to achieve the equivalent of a Moore's curve for software productivity relying on technology not evolved in a production environment.

Reusability. Reuse is critical as a major justification for object-oriented programming (OOP). Unfortunately, there is no accepted definition of reuse nor a measure of its achievement.

One can take the view that reuse only has meaning when functionality is inherited as defined in the OOP sense. One then "bends the metal" around the reused module (class) to accommodate the changes. We call this "reusability in the OOP sense." One must consider the resulting modification effort and the possibility of inheriting functionality one does not want. If visibility into what one is inheriting is low, conflicts can arise downstream.

Our concern is that measuring the effort required to reuse a software module in a new function. We want to minimize the energy spent in development and support. This leads to a practical definition of reusability as the reduction in effort when one starts with a previous module and modifies it to produce the new function. The amount of modification is subordinate to the energy required to build and support resulting modules. Given a development environment that minimizes this energy, we can expect higher productivity.

Consider reuse in a production environment. Given we can copy a module and modify it, the relative amount of change affects our approach. If we modify a significant percentage of the module, then supporting two distinct modules is difficult to argue. On the other hand, for large complex modules, one may find the percentage change quite small. In these cases, the original module is usually composed or submodules, most of which remain unchanged. The unchanged submodules can become utilities that remain intact to support both higher-level modules.

/ to see these modules just ... can see their modules. This implies visualization of the architecture, a property that has no counterpart in OOP

Product Measures

Before addressing measures for comparing software development environments, we consider measures of the end product under development.

Functionality. Software systems are serving an ever-widening range of functionality. WTien comparing software development environments, one must evaluate their ability to handle the increasing functionality.

Poorly specified requirements are often cited as the cause for late and buggy software. Sometimes this is true. However, the authors are aware of multiple cases where functionality was well specified, including user-supplied test data, to determine whether requirements were met, and the software efforts still failed.

A more important factor appears to be the amount of functionality one must deal with. We need to be able to quantify the size and complexity of the function space specified for a software product in order to determine the difficulty one faces in development and support for that product. This has been addressed in the function-point method. Measures of this type can be quantified for benchmark experiments.

Complexity. Information about the number of functions built into software is likely to be insufficient when trying to predict levels of difficulty. Productivity can take on significant variations due to different levels of complexity of the functions. The level of complexity of each function must be considered when measuring the difficulty in developing a piece of software. Complexity factors can be quantified for benchmark experiments.

Quality. As functionality and complexity grow, the number of opportunities for bugs multiplies. Knowing that two pieces of software have equal numbers of new bugs found per month is not sufficient to determine the comparative quality of each. One may have much more functionality than the other. Furthermore, one may have much heavier usage than the other. These factors must be accounted for when comparing the quality of different pieces of software.

Quality of software can be measured in terms of the availability of its specified functions; the time and cost to support that software to maintain an acceptable level of availability, which must be determined by the users of that software. Measures of availability can incorporate the level-of-usage factor. Here, we assume software is designed to meet a quantified level of quality.

Productivity Measurement Considerations

The accompanying figure depicts two characteristics that can be used to derive a measure of the productivity of a software development environment. The top characteristic shows an investment curve for development and support of software. The area under the curve represents the product of time and cost per unit time, yielding the total dollar investment to build and support a piece of software. More time and money is spent supporting product enhancements and error corrections than in development.

The second characteristic illustrates the revenues generated from product sales and maintenance fees per unit time. Revenues start to flow when an Initial Operational Capability (IOC) is reached, and cease upon System OBsolescence (SOB).

If the development time (to IOC) is stretched, and total development costs remain constant, that is, the expenditure rate is slower, then the time to reach revenue growth is pushed out. Total revenues can be reduced if competition enters earlier or if the product becomes obsolete. This causes loss of Return On Investment (ROI = total revenue - total investment). This can happen if initial product quality is not sufficiently high since customer dissatisfaction will inhibit sales growth and encourage competition.

Improvements in productivity must be reflected in improvements in ROI. Therefore, productivity is inversely proportional to the costs incurred. This comprises development costs and support costs. Additionally, if development costs remain fixed while IOC is reached in half the time with equal quality, revenues can start flowing earlier. This implies that if developer A spends money twice as fast as developer B, but reaches the same quality at IOC in half the time, A can expect a higher ROI. It takes much higher productivity for A to achieve this.

Ifm^ is man-hours spent in time period i, then total cost, C, can be estimated to be proportional to

C = K*I m^ = K*M,

where M is the total man-hours expended during development and integration, and K is a constant that depends on overhead and general and administrative expenses. We note this only reflects the cost part of productivity. If time to complete the project, T, is factored in directly, then productivity can be inversely proportional to

C-T = K-M*T.

We are not stating this is the measure of productivity. We are asserting that one must conduct experiments and take measurements to validate such an hypothesis. We encourage other hypotheses; but whatever the measure, it must be supported by valid repeatable experiments. We note that the support mode is typically dominated by incremental developments (enhancements), and can be treated accordingly. We also note that, if a given level of quality is achieved for competing software systems, then the revenue side is accounted for fairly, since other factors, (for example, marketing costs), are neutralized.

Factors Affecting Productivity

Here, we offer the properties of a software development environment that have been known to affect the man-hours and time to develop and support a software product.

Independence. Since reuse of modules has been a stated objective in the OOP revolution, we start with reuse. When attempting to reuse a module, one must be concerned with the independence of that module relative to its use by different higher-level modules. If it is not in the same task, then one may have to copy it, for example, as a library module, for use in different directories or platforms. If it needs other modules to operate, they must also be copied.

The more a module shares data with other modules in a system, the higher its connectivity to other parts of a system. The number of connections is measurable.

The higher the connectivity, the lower the independence. When designing hardware modules to be independent, one works to reduce the number of connections to other modules to a minimum. We note that visualization of the architecture is critical to performing this design task.

WTien building software using OOP, class abstractions cloud the ability to visualize connections. Understanding how data is shared between software modules can be difficult, especially when inheriting classes that inherit other classes. It is difficult to inspect a module to determine its degree of connectivity and understand the way it interacts with other parts of the system. Hiding and abstraction in the OOP sense makes it difficult to simply pull (copy) a module from one system and place (reuse) it in another. This difficulty in reuse as defined by OOP stems from the effort required to determine the level of independence.

In the case of hardware, one designs for minimum connections between modules. Using CAD tools provides a visualization of the architecture. Connectivity (coupling) is a key property affecting design productivity. This is true in software as well. But to fully understand this principle, one must be able to really see the architecture.

We take for granted our ability to compare computer hardware productivity using benchmarks and purchase hardware based upon them. There are no acceptable productivity benchmarks/or a software environment. Comparisons are generally based upon literature advocating a given method. Invariably they lack scientific data to support the claims.

Understcmdability. When managing a large software project, one gets to witness the loss of energy that occurs as two programmers reinvent the same module. Energy is also lost trying to decrypt algorithms coded to minimize the number of keystrokes used to write them, or to maximize "economy of expression."

If these algorithms are passed on to others, they may become enveloped in comments that explain the code, sometimes multiplying the size of a listing by whole numbers. Some claim that understandability of a language can be gauged by the average number of comments in the code. Understanding the code is a major factor in software productivity, especially in the support phase of the life cycle of a product.

More important than language is the underlying architecture of a system. This property is difficult to fathom if you have never seen a direct visualization of software architecture. This is only accomplished if there is a one-to-one mapping from drawings of the architecture to the physical layer, such as the code, just as there is in a drawing of a complex computer chip. We believe that without this visualization, significant productivity improvements can never be achieved for software.

The opposite situation occurs using OOP. The architecture is hidden behind the code—the only visual representation of the real system. Diagrammatic representations are abstractions and do not reveal the true complexity or hidden dependencies.

Understandability of the architecture also contributes to the design of independent modules. We believe one can measure the degree to which one can visualize software architecture and relate that to productivity.

Flexibility. One motivation behind the eXtreme programming movement, as well as Microsoft's approach to software, is the incremental approach to software, where functionality is added in small pieces, often with a working "daily build."

CAD tools make hardware architectural changes easy, especially when a system has been designed on a modular basis. A CAD system that does the same for software, that is, starts with a visualization of the architecture on a modular basis and provides a one-to-one mapping into the detailed code, can ensure design independence of modules while allowing visibility of the desired details. This can provide real reusability.

With such a flexible facility, one can design a little, build a little, and test a little, growing a system incrementally to ensure components are meeting specifications and showing near-term results. One can quickly detect when changes cause components to fall out of specification ranges. Fault isolation is much more easily accommodated. These factors should all lead to higher productivity.

Visibility. Electronic circuits are described by systems of differential equations. Yet, it is difficult to imagine designers working without diagrams of circuits. As circuits get large, it is the architecture—the parsing of functions into iconic modules and lines to show how they are interconnected—that becomes overwhelmingly important. Visualization of the architecture is the key to productivity.

We claim this is also true with software. However, a one-to-one mapping from the architecture diagram to the code must be achieved in order to gain the benefits derived from the equivalent in hardware. This is only achievable when data is separated from instructions, as in VisiSoft, a CAD product. If there is a silver bullet in software, this is it. Productivity gains can multiply using this CAD technology so as to achieve the equivalent of a Moore's curve.

Abstraction. No one can argue the usefulness of abstraction. It certainly can help get through major design problems. It can also sever ties to reality in a production environment. It is easy to draw block diagrams for sequential tasks that relate to the code. But in highly interactive systems, mouse or keyboard event handlers support many functions, and the software architecture becomes orthogonal to user functionality. Block diagrams lose meaning when one faces a software design from the extremities of the functional interface to the detailed code.

Apparently, benchmark comparisons of different approaches to developing software do not exist because of the size of experiments envisioned to perform the task. People envision two teams developing a sufficiently complex piece of software using competing environments. One can see why such an expensive undertaking is not done.

But most experiments in science are not very large in scope. Focus is usually on creating sequences of small experiments that can lead to larger conclusions. We believe software should be broken into pieces such that the methods that produce them, including integration, can be examined experimentally.

Conclusion

As the quote from Kelvin implies, we cannot expect to improve software productivity without measuring it. The measures of a software product—functionality, complexity, and quality—are not new. They form the foundation for measuring productivity.

If a given level of quality is achieved for the same software system by competing organizations, their relative productivities can be measured as being inversely proportional to the product of their cost and development time.

Productivity of a software environment depends upon the understandability and independence of modules produced. These are inherent properties of a software development environment, and can be increased or decreased by design. Visualization and CAD techniques can improve these properties just as they do for hardware designers.

Finally, we must be able to measure productivity changes to validate our assumptions regarding its dependence on these properties. We believe this can be done using a large number of small experiments that, combined statistically, will represent the productivity of a development environment. We perceive the experiments are suitable for university collaboration. Q

DONALD ANSELMO (dranselmo@usweat.net) was the director of the
Computer Development Laboratory at AT&T Bell Laboratories responsible for Unix computer development. He is now an independent consultant.

HENRY LEDGARD (ledgard@eng.utoledo.edu) is a professor in the Department of Electrical Engineering and Computer Science at the University of Toledo, Ohio.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© 2003 ACM 0002-0782/03/1100 $5.00