Scan without Illustrations of Chapter 14 from Gilb: Principles of Software Engineering Management, Addision Wesley 1988, in 18th Printing 2004
Questions tom@gilb.com, www.gilb.com

Chapter 14

THE MANAGEMENT OF SOFTWARE PRODUCTIVITY

14.0 Productivity of results, not process.

Productivity should be measured in terms of net real effects on high-level management goals of a business or institution. Any attempt to quantify productivity by many common, but more partial measures, such as " volume of work produced " are a great deal less useful. These partial measures do, however, have a place. They can provide some insight and control over the productivity question in the early stages or at a low cost of measurement.

Productivity should be measured by the net effect of a solution on results. This means that we have to account for the cost of developing and operating the solution in both the short and long term, as well as the cost of all the side effects of the solution.

Productivity planning must be carried out at a high management level in order to guarantee the relevance of the solutions to management objectives. Productivity goals are usually multidimensional and complex, but they can be written down, agreed upon, and expressed in clear and measurable ways.

The tools for improving software productivity are many. They can be implemented immediately with interesting results, and then strengthened by a long-term evolutionary series of changes and improvements. Each of these changes is based on continual monitoring of productivity results up to that point.

You will find this chapter's main ideas summarized in the principles at the end.

14.1 What is Software?

Most professionals interpret the term "software" itself in a dangerously narrow way. Behind most uses of the term "software" we find the concept of what I prefer to call "logicware" - or what we call "programs".

Websters Unabridged Dictionary defines "software" as "the programs, data, routines, etc. for use in a digital computer, as distinguished from the physical components (hardware)."

Since the production of software today involves many more additional non-hardware components than were formally recognized in the early days of digital computers, it is only natural that we update our concept of "software" by including these new items in our consideration of software productivity. We cannot discuss "software productivity" adequately, if we do not have a complete definition of the term "software" itself.

I divide software into the following main categories:

- logicware (computer program logic)
- dataware (computer-readable files and databases)
- peopleware (plans and methods for organizing people to make use of the system or to develop it or test it)
- userware (user documentation in paper or display screen versions, and user command languages).

14.2 Evaluating the Software Product

Productivity is, as mentioned earlier, measured in terms of the planned attributes of the product. It is these attributes which will enable us to determine whether, and to what degree, the user has attained his objectives (user productivity). One "user" of the product can be the producer themselves - and the use can be to sell the product or to sell related products (such as hardware).

There are a large number of attributes which together determine the total short-term and long-term usefulness of software. They have been discussed extensively in this book - and are catalogued in Ch 19, "Templates". An extensive catalogue of software attributes was given in "Software Metrics" book, referenced earlierGILB-SM.

There are some software product attributes which are of immediate everyday value; for example reliability, usability, and work-capacity. It is productive work which is necessary to achieve the needed levels of these attributes. I mention this only because it is a very common failing to ignore these qualities, and to think that productivity is in "coding" the bare functional logic only. The result is an illusion of productivity, but not the reality.

It is a very dangerous illusion, since high quality attribute levels can cost the largest part of the entire development effort. This is easily illustrated by observing the huge effort needed to build extreme ease of use (usability) into software. The Apple Macintosh design effort is an example of this effortByte-MAC-84 .

14.3 The Long Term Productivity Considerations

Developer (producer) productivity produces good software effectively. User productivity is enhanced by the use of good software. The quality attributes of software impinge on user productivity.

The particular quality attributes which impact the productivity of both user and producer in the long run can be difficult to see. The primary ones are "Maintainability", "Extendability" and "Portability" (see "Templates" Ch. 19 for definitions) which are all related to the ease of change of the product in order to meet long term future needs.

If these attributes are poorly engineered in the software product, then there is a great danger that the product will die or become poorer in use. The investment needed to design and build these long-term qualities into the system will determine whether it is really productive in the future.

Many a software project has suffered from insufficient effort in the engineering of these areas - due to poor management leadership. They have created the illusion of software productivity (in the short term), at the expense of the long term productivity .

Somebody (it is not likely to be a programmer) who cares about the true long term productivity of the software effort, must ensure that these long-range factors are engineered into the software product.

You should not wait to be asked, because the marketing people and end users may not be wise or mature enough to explicitly ask for these properties. A responsible professional will raise the issue, and force the people requesting the software to include high quality long term attributes, or at least to take full responsiblity for not having done so.

THE USER AS JUDGE PRINCIPLE
The end users themselves should be the final judge of productivity in the sense of software quality, not the producers.

The intention of such principlesGILB-DP-83 is to ensure that we can measure the true user productivity given by the software product, in all important areas, throughout its lifetime. Here is a more detailed background for these principles.

14.4 Users should judge software - The BHP and Volvo Cases.

For software producers selling to a free market, there is adequate public judgement of the software quality in the trade press, by the sales statistics, or at user group meeting.For more captive users of software, such as those from a company producing software for internal consumption, a more drastic remedy is needed.

Volvo of Sweden provided this by making it mandatory for internal Volvo computer users to ask for a bid from their internal Data Processing development facility,while at the same time encouraging those users to ask for and accept alternative bids for better software products from outside suppliers.

One of the most interesting examples of a powerful internal control by the user of application software was at BHP (Broken Hill Pty.) , the largest Australian industrial corporation (steel, mining, oil, finance) from 1972.

The users seemed to hold total power over the software producers. After nine previous years of unprofitable and unresponsive data processing development, top management stepped in and introduced a user-controlled profitability measure of the software value. This applied to internal developments, as well as any support software required from outside .The result was that the "academics" fled, and the survivors became dramatically more responsive to the users' needs.

The basic mechanism was a continuous (monthly) application-lifetime budgeting and accounting system which compared a user-determined application "value" (in terms of real money savings or productivity increases - no "intangibles") to the real current costs of running the application. Projects which fell below a minimum set level of profitability were initially given a chance to improve the ratio. If this failed, they were quickly killed.

The net result, even in the first year, was that in spite of a budgeted loss of several hundreds of thousands of dollars, the actual result was a clear profit of several hundred thousand dollars for the surviving software applications.

Nobody in BHP was worried about producing "lines of code". The entire surviving data processing staff (six hundred people) had only two questions in their minds about all projects, at all times.

- how can we keep the costs down as low as possible?

and

- how can we make the software so useful in terms of user cost saving and user productivity (more steel plant productivie capacity for example) that the user management profit centers will give our product a high dollar rating (part of which is charged back to them), and thus keep it alive?

14.5 Continuous Monitoring

THE NEVER ENDING JUDGEMENT PRINCIPLE
Software systems need to be judged on a continuous basis throughout their lifetime - not just by the first user, the first month.

Software applications cannot simply be judged once - in a post-implementation return-on-investment-analysis (though in my experience, even this is not done often enough).

Here are some of the reasons why the evaluation of software applications should be reviewed regularly:

- hardware costs change dramatically year by year.
- maintenance changes might degrade performance and other qualities
- the user-environment changes - yesterday's winner may be tommorrow's loser.
- management employees change jobs, and with that goes a style of management which may have been key to the value of the product.

14.6 Formal Testing of Productivity-Related Software Attributes

THE MULTIPLE TEST PRINCIPLE
Software systems should have formally defined acceptance test criteria which are applicable at all times for all critical qualities.

Several software qualities (for example maintainablity, portability, and usability) are keys for allowing the product to be really productive. All of them are measurable and testable in practice (see "Templates" Ch.19 and Software Metrics GILB-SM). There are unfortunately far too few software professionals who know anything about measuring and testing these properties of software.

Software engineering management must institute a rigid requirement for testing of

these qualities and other critical attributes of the software system. If they fall below critical levels, as determined by yourselves and your users, it could kill the entire software effort or product.

14.7 Productivity is Managerial not Technical.

THE PRINCIPLE OF SOFTWARE PRODUCTIVITY:
It is not the software itself which is productive. The interesting results are created by people who make use of the software.

Most of the productivity improvement techniques with really significant impact are "managerial", not "technical" in nature. This was the conclusion drawn by Horst Remus of IBM after years of monitoring productivity figugures at IBM at their California Santa Teresa LabsRemus-80 . My own observation, based on measures of software project productivity is the same.

Many software technologists seem totally ignorant of the existence of the managerial and organizational methods which lead to highly improved human productivity. The technologists seem to believe that productivity is to be had through technical means, such as ever more sophisticated programming languages, or more sophisticated software support for their working environment. There is some truth in this viewpoint, but it is not where the really big improvements have been found.

This point is brought out in a number of management textsPeters-PFE . It is clearly motivation and organization that increases human productivity in relevant directions. Technical devices may increase productivity "in the wrong direction". (We can always increase "lines of code" - even where the software being produced for the market is the wrong design!)

14.8 Management Productivity

Productivity of management at all levels above the software technologist can be improved by:

- concentrating on determining user requirements,
- particularly noting those fluctuating or uncertain user requirements which will require a suitable flexible softecture (software architecture).
- creating an organization which is totally user-result-oriented, even at the most technical level.
- implementing measurement systems which relate all technical work to corresponding user-value and user-cost concepts.
- filtering user needs through competent business analysts, infotects, softects and software engineers (do not allow things to go directly to the softcrafters).
- provide users with the means to do a maximum of "software development" themselves; either by building such devices into the product, or by supplying user-oriented development languages (like spread sheet software) to the users.

14.9 Professional Productivity

The "business analyst" function can increase productivity of the user by:

- avoiding computerization when other options are better or more cost effective,
- worrying about the "non-software" aspects of making your software be productive for the user (like whether people are still motivated to use it at all).

The business analyst operates at a higher level than most present day system analysts. Too many analysts are primarily concerned with analyzing the function to be automated. The business analyst does not even presume that software is to be written, or even that there is an information system problem.

The "infotect" can contribute to professional productivity by making sure that the information system problem is channelled to the best solution area.

Too many analysts are trained and working in an environment where they really see only one technical solution; for example, the company standard computer and prevalent languages and database support system.

Sometimes a computer is not the most cost-effective way of doing things, and some alternative computerized solutions are far better than the conventional one. The infotect is charged with finding the most productive "results" solution, irrespective of the devices need to accomplish it.

The "softect" is a necessary function in a large software engineering environment, in which there are many specialist software engineers. The softect is the necessary synchronization and co-ordination function for the many specialized engineers and builders. The softect presumes that software must be designed, and is only concerned with finding a technical solution set which will satisfy the multiple conflicting objectives of the user as well as possible.

The "software engineer" is also a productivity professional. Currently I find that we speak of software engineering as though it were a single speciality. But the history of other professions makes it clear that specialization is the norm for large projects. We can certainly identify the specialists even today in this area, even though they do not always call themselves "software engineers".

The softect is also a specialist software engineer, the speciality being overall control of a complex engineering process. Other software-engineering specialists are, for example, concerned with work-capacity, availability, usability and security.

Software engineers can be expected to increase productivity in their special area of competence. That is exactly what their training should enable them to do. One measure of their competence is how much they can improve their specialty attributes; another is the degree to which they can correctly predict or estimate what they will in fact achieve when all side-effects are considered.

14.10 Productivity Tools

Most all of the highly-touted productivity tools (programming languages, software support environments, database support systems, operating systems) offered by traditional industry, have failed to deliver substantial net user-productivity in a well-documented way. This has not prevented them from claiming impressive productivity increases, forgetting that the real end-product is user productivity. My experience in years of trying to substantiate such claims is that:

- they are based on isolated cases and may well be due to uncontrolled factors (the super-programmer on one project, for example),
- they do not note, or even consider, undesirable side effects (such as performance destruction, or portability reduction) which need to be considered in any fair evaluation of real productivity.
- almost none of them meet the conditions of scientific verification via controlled experiments, and statistically valid assertions.
- most of them are concerned with producing only one area of "productivity", namely "logic for functions". Few of them address any of the critical attribute dimensions of technical software quality and cost, even fewer address user benefits or results.

I do not deny that some of these productivity tools have a beneficial effect. But I have not yet found evidence for impressive net benefits in software productivity which are as impressive as those I have found for methods such as Fagan's inspection, for evolutionary delivery and even the simple act of formal specification of objectives.

14.11 Fagan's Inspection Method

Fagan's Inspection methodFagan-76 has regularly measured net productivity increases of about 25% to 35% in software project time to delivery. Exceptionally high savings have been reported in the test planning areaLARSON-75. Larson reported, with Fagan later confirming the long term consistency of the effect, eighty-five percent of test effort was saved as a result of using Inspection to check the quality of test design and planning. My own clients have publicly reported 10 to 1 (ICI UK on 400 of 800 programs), 18 to 1 and 30 to 1 improvements in maintenance effort needed for software which has been inspectedCROSSMAN-79 .

These are the once-off productivity effects of inspection. The really significant news about inspection is that the statistical feedback it gives on defects and costs provides the manager with a software engineering management accounting system. This can be used to identify a wide range of productivity problems in a software developement process, and then to measure and see if the suggested solutions are working as expected.

Both IBM in the US and ICL (International Computers)Kitchenham have regularly used Inspection for monitoring and improving their software development processes, in order to improve productivity.

The illustration "Inspections Long-term Effect", from IBM Santa Teresa Labs in California, shows the cumulative raw productivity achieved in terms of code produced per work-month.

The real productivity benefit is greater than is indicated by the "Productivity" curve alone. At the same time, a quality indicator (lower defects) is improving, and this saves productive effort in error repair (maintenance cost) , as well as enhances the desireability of the suppliers products to customers. It is highly probable, because of the nature of Inspection, that other quality indicators are also increasing the net productivity of the use of inspection, as a management accounting system, at the same time.

14.12 The Productivity of Evolutionary Delivery

The most impressive practical method for ensuring dramatic productivity in software projects, is still the least understood of all the methods - evolutionary system delivery. (see details in chs. 13 and 15.)

IBM FSD is a long time leader in the use of this method in the software engineering arena (since about 1970)Mills-80 . Mills reports that all projects using the method for the last four years have been completed "on time and under budget". Surely that is a form of productivity in itself which few software engineering managers can claimGILB-EVO-85 .

14.13 Project Data Collection and Analysis

Another under-utilized method for productivity through management analysis of facts is the use of systematic project data collection and analysis.

The only really good example, in terms of an ongoing collection process, I have found in the public literature of this is at IBM Federal Systems DivisionFelix-Walston. An interesting collection of data, but not so clearly ongoing, is published in Boehms "Software Engineering Economics.Boehm-SEE-81 Many pages of project data are collected at the end of each project and analyzed in an APL database at IBM Federal Systems Division, Bethesda, Maryland.

IBM FSD is able to systematically compare a large number of projects on a numer of factors regarding cost, delays and methods used. This enables them to spot methods or environments which are more or less productive, and to take management action to weed out the bad and to nurture the good.

Most software engineering environments are not able to do this anywhere nearly as well. Most rely on the faulty memories of old warriors. The objective of software engineering management is to increase the predictability in meeting our objectives, whatever those objectives may be. We can therefore measure our ability by measuring the deviation from our plans in high priority areas.

We must do this statistically, by collecting the kind of data which IBM Federal Systems Division has been collecting, or which Barry Boehm of TRW Systems has collected (see above references). For example Boehm (in Software Engineering Economics) says that in his selection of projects, 70% of the projects would be within 20% of the cost predicted by his COCOMO cost estimation model, and 30% of the projects would be outside that estimated deviation.

Harlan Mills of IBM claims to have found a method, in the same class of systems that Barry Boehm is dealing with, which guarantees no significant negative deviation for two important attributes (delivery on schedule and cost). By the above principle, Mill's methods (Evolutionary Delivery) are better software engineering management principles, in terms of getting real management control over cost and delivery, than using the best known cost estimation models. Both examples are based on comparable sets of statistics for comparable projects.

I suggest that this example shows a fair and useful way to judge methods of software engineering management.

14.14 Summary

We can sum up with a set of principles regarding people productivity.

1. IF YOU CAN'T DEFINE IT, YOU CAN'T CONTROL IT.
The more precisely you can specify and measure your particular concept of productivity, the more likely you are to get practical and economic control over it.

2. PRODUCTIVITY IS A MANAGEMENT RESPONSABILITY.
If productivity is too low, managers are always to blame - never the producers.

3. PRODUCTIVITY MUST BE PROJECT-DEFINED; THERE IS NO UNIVERSAL MEASURE.
Real productivity is giving end users the results they need - and different users have different result priorities, so productivity must be user-defined.

4. ARCHITECTURE CHANGE GIVES THE GREATEST PRODUCTIVITY CHANGE.
The most dramatic productivity changes result from radical change to the solution architecture, rather than just working harder or more effectively.

5. DESIGN-TO-COST IS AN ALTERNATIVE TO PRODUCTIVITY INCREASES
You can usually re-engineer the solution so that it will fit within your most limited resources. This may be easier than finding ways to improve the productivity of people working on the current solution.

6. A STITCH IN TIME SAVES NINE
Frequent and early result-measurements during development will prevent irrelevant production.

7. THE OUNCE OF PREVENTION- (WHICH IS WORTH A POUND OF CURE).
Early design quality control is at least an order of magnitude more productive than later product testing. This is because repair costs explode cancerously.

8. DO THE JUICY BITS FIRST.
There will never be enough well-qualified professionals, so you must have efficient selection rules for sub-tasks, so that the most important ones get done first.

Byte-MAC-84
For insights see BYTE magazine, Feb 1984 (p 58-80), Aug 1984 (p.238-251), Dec 1984 as well as "MacWorld" magazine Vol. 1 No. 2 (May June 1984) p. 122-127 and "The International Mac.", for a series of articles relating to the effort to design ease of use into the Macintosh.
GILB-DP-83: Gilb, T., "Increasing Software Productivity", in Data Processing (UK), p 16-20, Vol. 25, No. 7, September 1983. This is the original statement of these principles and the initial draft of major ideas in this chapter.
GILB-SM op cit.
Remus-80 : Remus, Horst. Planning and Measuring Program Implementation. IBM Technical Report TR 03.095 (June 1980). Remus has many other related reports publicly available.
Peters-PFE: Peters, Tom and Austin, Nancy. A Passion for Excellence. 1985 Random

House(USA) and Collins (UK) ISBN 0-00-217529-0. 437 pp.
Fagan-76: Fagan, M. E. Design and Code Ispection to Reduce Errors in Program Development, IBM SJ Vol 15, No. 3, 1976.
LARSON-75: Larson, R., Test Plan and Test Case Inspection, IBM Technical Report TR 21.586 Kingston NY, April 4, 1975
CROSSMAN-79: Crossman, T. ,Some Experiences in the Use of Inspection Teams in Application Development, Guide/Share Applications Development Symposium Proceedings, Montery CA, Oct 1979.
Kitchenham : Kitchenham, B. see frequent (1982-85) contributions to ICL Technical Journal by this author for reports based on Inspection data. ICL Bridge House, Putney, London SW6 3JH, GB.
Mills-80
Mills, Dyer and Quinnan articles in IBM SJ, Vol 19, no 4
GILB-EVO-85: Gilb. T., Evolutionary Delivery Vs. the Waterfall Model, p. 49-61, ACM Software Engineering Notes, July 1985.
Felix-Walston
Felix, C. P. and Walston C. E., A Method of Programming Measurement and Estimation, IBM SJ Vol. 16 No. 1 (1977) pp 54-73. The 14 page questionaire is available on request from Walter Ellis, IBM FSD, Bethesda Md.
Boehm-SEE-81Boehm, B. W., Software Engineering Economics, Prentice-Hall, 1981,, 767 pages, ISBN 0-13-822122-7