

WHAT IS YOUR SOFTWARE WORTH?

By applying well-known principles of intellectual property valuation, sales expectations, growth of maintained software, discounting to present value, and the like, a method is presented for valuing software based on the income that use of the software is expected to generate in the future.

BY GIO WIEDERHOLD

There exists voluminous literature on cost estimations for producing software, but that literature largely ignores the benefits of *using* that software [4]. Even software engineering management approaches termed “earned value management” only deal with expenses within a development schedule [1]. While software creators believe their product is valuable, they rarely quantify its benefits. Much investment in software engineering has been motivated by military and governmental applications, where benefits are indeed difficult to quantify. When benefits of software in commerce must be quantified, it is typically left to lawyers, economists, software vendors, or promoters to assign value to our products. The results are mostly inconsistent [8].

Why should software creators care? In many other fields the creators have a substantial awareness of the value of their products. Architects are aware of the market for houses they design and builders of bicycles will know their market. But since software, once written, is easy to replicate at a negligible cost, each subsequent instance is sold for much more than its incremental cost. Potential sales volume is a key factor required to set a price allowing an adequate future income. That income determines the value of the product to the creator.

ILLUSTRATION BY PAUL WILEY

The value of software to a purchaser is independent of the cost and effort spent to create it. A few brilliant lines of prose or code can have a very high value, whereas a million lines of code that generate a report that nobody reads have little value. If creators are aware of the potential value of the product they will be better prepared to make decisions on product design and the effort to be expended. The motivation for this article is to increase the awareness of the computing community how the result of their work may be valued. That should, in turn, affect how software engineering is practiced.

I first present the principles of valuing intellectual property (IP), based on the income generated by a software product. The valuation addresses software as it exists at some point in time, and ignores the cost of its creation. Once the value of the product is estimated, one can compare that value with the cost of its creation and decide if the overall project is profitable, or, if it seems not to be, what must be changed to make it so.

Software, since it improves and grows over time, presents novel issues not seen in other intangibles as music and books. Maintenance to sustain software effectiveness occurs throughout the time the software is in use; that is, while the software actually generates benefits. Maintenance costs over time typically exceed the original software development cost by factors of 2 to 10. Maintenance causes growth of software, and we show here how three types of maintenance efforts together affect growth and value. The growth of software will also be modeled using some rules-of-thumb derived from experience.

After the discussion on the diminution of software IP we predict sales volumes for software. Standard business methods are presented, selected for their suitability in modeling software sales. We then combine prior results to compute the value for the example. To illustrate the use of a quantitative model for analyzing alternatives, we include maintenance income, representing a service-oriented business model. We conclude with some advice for individuals, managers, and educators.

The goal of this article is to bring together information from domains that rarely interact directly: software engineering, economics, business practice, and legal sources.

PRINCIPLES OF IP VALUATION

Assigning value to intangible property is assuming greater and greater importance, as our society moves from dependence on hard, tangible goods to a world

where knowledge and talent creates the intangible goods we need and desire. Many approaches for valuation compete [5]. Tangible goods are produced by a combination of labor, capital, machines, and management; the quality of the human components plays a minor role in valuing such a company. Even today, the book value of a company given in its annual report is merely the value of its facilities, inventory, equipment, and finances.

That book value has little to do with how investors value companies in the software domain. For example, we obtain SAP's annual report for 2003 and learn that its book value (assets - money owed) was about €6.3 billion. But using the market price of the shares (about €100) and the number of shares in circulation, about 315 million, we find that SAP's shareholders valued the company at €31.5 billion. The investors in SAP base the value mainly on the income they expect to obtain over time from their shares. The difference, €25.2B, is due to the intangible property owned by SAP and its shareholders. The value of the SAP brand in 2004 was estimated at €6.8B. Software, and the knowledge of how to produce, improve, and sell it, comprise the rest, €18.4B. U.S. high-tech companies show similar ratios, but no routine report provides valuations of their intangibles [11].

The intangible property owned by a company in the knowledge-based domain includes the technical knowledge of its staff, the competence and insights of its sales force, the business knowledge of its management, the worth of its trademark, its reputation, and the value of its software inventory.

The reputation of a software company can be increased by advertising and the value of its software can grow by research and development. These two components are the considered IP of a business. The work force of a company cannot be considered a property. Valuation of IP is required when software product lines are purchased or transferred. When entire companies change hands, then the work force is assigned a value as well. Although software is the most tangible of the IP owned by businesses, its valuation is still poorly understood, leading to a gamut of problems [8].

The value of software IP. Investors in a software enterprise assert through their stock purchases that

IP RULE: THE VALUE OF THE INTELLECTUAL PROPERTY IS THE INCOME IT GENERATES OVER TIME.

That simple rule is the basis for any IP valuation. Estimating that future income and reducing it to a single current value is the task to be undertaken [11].

This article focuses only on software, likely the

largest IP component owned by companies involved with computing. Ownership of software is not limited to companies that produce software as a product. The majority of current businesses creates, purchases, maintains, and benefits from software. Banks cannot function without software; there is no other way to determine what is due to a customer or what the customer owes. Manufacturers cannot live without software; the designing process, obtaining and allocating resources, managing the workflow, and shipping the goods out all depend on software—and companies that exploit software better will be more profitable.

Estimating income. To value the IP inherent in software, one must estimate how much income the software will generate during its future life, which also requires estimating its life. We distinguish here software producers and software users.

If the software produced is sold to others, the expected income depends on the sales revenue, the product of the amount of software sales, and its price. We assess the software from the viewpoint of the seller. When a new version of a software product has been prepared and is ready for sale, sales of the prior version will rapidly diminish. Since the costs of copying and packaging software are very low, there is no benefit in continuing to sell the old software, a characteristic particular to intangible property. Software differs from other intangibles: while a book written and printed two years ago can be profitably sold for, say 80% of its new price, selling a prior version of software at an 80% price makes no sense for the seller. Supporting old versions is only a cost for the seller, while being out-of-sync creates inefficiencies for the customer.

Furthermore, the new version will be better than the previous version. For the purchaser obtaining cheap, but obsolete software creates expenses for adaptation and integration. A new version of the software product includes much of the code and all of the functionality of the prior version. Disasters have occurred when new versions did not replicate all prior functionalities [12].

If software is developed and used internally, valuing

it as a distinct income-generating investment becomes impossible. Today, it is rare that a broad set of new software applications will be installed within an ongoing company. A company employs many diverse resources to generate its income. Income can still be assigned based on a belief that the management of a company is rational in the allocation of its resources—a standard textbook assumption.

PARETO RULE: AT THE OPTIMUM EACH INVESTMENT DOLLAR SPENT GENERATES THE SAME BENEFIT.

If a company spends more than optimal on software and less on people or marketing it would reduce its income, and vice versa. Given that rule, corporate net income created by diverse expenses can be allocated according to the proportion of costs incurred. The fraction spent on software from year to year will vary, but over its life such variations even out. If a company behaves irrationally in its spending, more so than its peers, it is bound to have lower net profits, and its IP and its stockholders will suffer as a result.

Income-based measures don't work in governmental and military settings. In those organizations measures of productivity and cost-avoidance must be combined to produce a surrogate for income. In those settings, and in other non-profit institutions, as academia, using an assumption of rational behavior for relative allocation is also questionable. Valuations of software will hence be quite inexact and mainly useful for making comparisons.

Revenue and gross profit. In business financial reporting the revenue realized is immediately reduced by the cost of the goods sold. The effort to make the first unit of a product is a major cost in both cases, but for software the manufacturing cost-of-goods is negligible. If software distribution is over the Internet, there are no incremental costs for each sale. Revenue and gross profit, the revenue after the cost-of-goods sold, become similar, and common financial indicators, as gross margin, are close to one and meaningless.

Since we only assess here the value of existing soft-

If creators are aware of the potential value of the product they will be better prepared to make decisions on product design and the effort to be spent.

ware, we ignore its initial research and development cost and its negligible manufacturing cost. Now the income per unit is equal to the revenue, that is, the price it fetches and the sales volume over its life. However, there will be ongoing costs to keep the software viable.

SUSTAINING SOFTWARE

Before we can aggregate the income generated by software in order to value its IP, we must consider what happens to software over the time it generates income. It is here where software differs crucially from other intangible goods. Books and music recordings remain invariant during their life, but software keeps on changing.

Methods used to depreciate tangibles as well as intangibles are based on the assumption that the goods being valued lose value over time. Such depreciation schedules are based on wear, the loss of value due to obsolescence, or changes in customer preferences. However, well-maintained software, in active use, does not wear out, and is likely to gain value [12].

All substantial business software must be sustained through ongoing maintenance to remain functional. What maintenance provides was stated many years ago by Barry Boehm [4]:

“.. The majority of software costs are incurred during the period after the developed software is accepted. These costs are primarily due to software maintenance, which here refers both to the activities to preserve the software’s existing functionality and performance, and activities to increase its functionality and improve its performance throughout the life cycle.”

Successful software products have many versions, long lifetimes, and corresponding high maintenance cost ratios over their lifetime. Software lifetimes before complete product (not version) replacement is needed are typically 10 to 15 years, and are likely to increase [11]. Version frequency is determined by the rate of changes needed and the tolerance of users to dealing with upgrades.

Maintenance costs of such enterprise software amount to 60%–90% of total costs [10]. Military software is at the low end of the maintenance cost range, but is poorly maintained and instead requires periodic replacement. Its users can’t complain much.

Continuing improvement. We use IEEE standard definitions for the three classes of long-term maintenance. Collecting and responding to feedback, crucial to IP generation, is detailed next.

- *Corrective maintenance* is essential to keep customers. In practice, most required bug fixing is performed after software delivery. If it is not suc-

cessful, the product will not be accepted in the marketplace and not have any significant life.

- *Adaptive maintenance* is needed to satisfy external mandates. Adaptations allow the software to deal with new hardware, operating systems, network, browser updates, as well as other software used in the customers’ environment. Software must also be adapted to changing governmental and professional regulations.
- *Perfective maintenance* keeps the customer happy and loyal [2]. Perfecting includes performance upgrades, assuring scalability as demands grow, keeping interfaces smooth and consistent, and being able to fully exploit features of interoperating software by other vendors, as databases, Web services, schedulers, and the like. Perfecting makes existing software work better.

Bug fixing for software accepted in the market eventually reduces to less than 10% of the maintenance effort. Adaptation consumes 15%–50% of the maintenance costs. That effort needed varies with the number of interfaces that must be maintained. Perfecting software is known to require about half of maintenance costs in the long term. Marketing staff often touts the results of perfective maintenance as being novel and innovative, even when base functionality does not change.

The effectiveness of maintenance is greatly hindered by poor design and lack of adequate documentation [3]. Well-designed systems will be improved more rapidly, have a longer life, and hence, paradoxically, consume more maintenance [6]. Figure 1 depicts relative effort distribution over time; in practice the ratios will differ depending on the setting and on external events.

Good maintenance keeps existing customers happy and improves the product being sold. A high level of maintenance retains customers (it is easier to keep an old customer than to gain a new one) and gains income (good maintenance has a high value to the customer). Fees for maintenance can contribute in time more revenue than new sale licenses.

Sources of IP in the maintenance phase. Maintenance is too often viewed as work requiring little expertise. That view ignores the contribution of the diverse information sources needed for maintenance.

Corrective maintenance is based on feedback from the users and customers. Error reports will be logged and filtered so that only significant problems are forwarded to the engineering staff. Corrections are often needed when unexpected cases or novel combinations

of usage are identified. If the original developers are still involved, fixing bugs becomes an opportunity to learn, rather than a task to be dreaded. Code to deal with those cases is added, and so the software grows.

Adaptive maintenance is based on information from standards bodies, from hardware vendors, from software vendors who make products that interface with the software, and from government sources. Major suppliers have representatives at many external organizations and vendors. It is rare that prior interfaces and capabilities are removed, thus the software grows.

Perfecting software is a process, perfection is elusive. Perfecting requires the entire organization to listen, prioritize, and act on feedback. Open-minded sales staff can provide the best input [6]. For instance, no scientific method assures that interfaces with human users will be attractive; there is more variety of customers in the global market than any specification can encompass; and there will always be new uses for software that with some changes can be catered to.

Perfecting updates should not disturb current users. Perfecting software is a major contribution to growth.

The inputs that lead to maintenance actions represent IP not present in the original product. Hence, they should not be valued as part of the original IP.

GROWTH OF SOFTWARE

Since software IP is embedded in the code being used, and that body of code changes over time, we must now measure what happens to the code and its functionality.

Two metrics are in common use for estimating code size and effort: Lines-of-code and function points. Both have problems, but they also complement each other.

The size of programs can be measured once the software is completed. One can look at lines-of-source-code (SLoC) or the volume of binary code. Some adjustments are needed in either case. When counting SLoC comment lines should be ignored, and multiple statements on a line should be counted distinctly. Software modules that appear multiple

times are counted only once to obtain the IP. If modules have been written in multiple languages, then conversions must be made. The function-point (FP) literature provides tables that relate them to code for many languages [7]. Common procedural languages have ratios of about 100 SLoC to one FP, while Visual Basic is rated at 35 SLoC/FP, allowing normalization.

Binary code sizes require adjustments as well.

Relative value of old and new code. We measure code sizes to allocate its relative contribution to IP. The assumption we will make, namely that the value of a unit of the remaining code is just as valuable as a unit

of new code, simplifies the IP analysis in the next section. This metric provides a surrogate to track IP growth over time.

There are valid arguments that code size is not a surrogate for the IP contents. One argument is that later code, being more recent, represents more recent innovation, and hence should be valued higher. An argument in the opposite direction that the basic functional-

ity is represented by the initial code. There may have been a few lines of brilliant initial code, slowly buried in a mass of subsequent system interfaces and later tweaks and fixes, that are critical to the IP. The architectural component of software also represents valuable IP and changes little over the life of the software.

Much more code is inserted later to deal with error conditions that were not foreseen originally. Code that was added during maintenance has its value mainly in terms of providing smooth operation and a high degree of reliability. The original code provided the functionality that motivated the customer's acquisition in the first place. If that functionality would have been inadequate the customer would not move to any subsequent version. However, new code will include adaptations and perfections that motivate additional sales.

Given that the positives and negatives can balance each other out, it is reasonable to assign the same value to lines of old and new code to measure the IP contributions in software. As long as the methods to obtain metrics are used consistently and without bias, the numbers obtained will be adequate for the inherently difficult objective of valuing software.

Growth of code. The maintenance activities that sustain software cause the software to grow in size, as

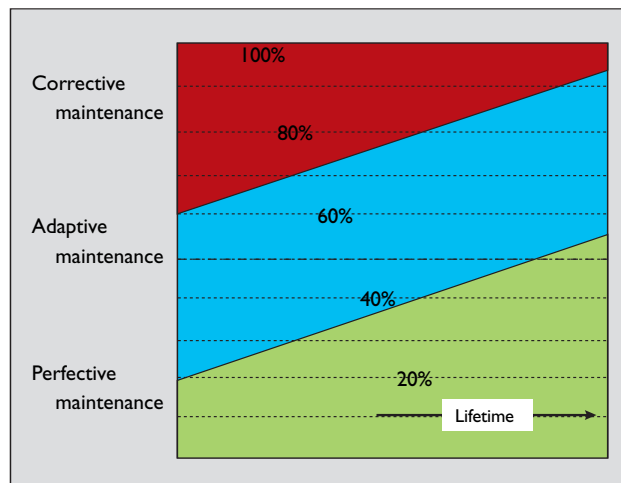


Figure 1. Maintenance effort over the lifetime of software.

previously discussed. Hennessy and Patterson presented a rule that software, in terms of lines-of-code, grows by a factor 1.5 to 2 every year. However, this rule implies exponential growth, expected for hardware, but such a rate of growth cannot be sustained in software development. Phil Bernstein of Microsoft has suggested that

PB RULE: A NEW VERSION OF A SOFTWARE PRODUCT SHOULD CONTAIN LESS THAN 30% NEW CODE.

Substantial growth implies too many code interactions in the new version, reducing reliability. Cost estimation tables support such barriers [7]. Complexity limits were clarified in Fred Brook's essays: since

code due to growth is 50% in Version 2, 33% in Version 3, 25% in Version 4. By Version 3 software growth obeys Bernstein's limit. In the early phases of a product a higher rate of growth is certainly possible; all the original developers are likely to be still on board, there is good understanding of the code, and feedback from the field can be rapidly accommodated, so exceeding the PB rule limit initially seems acceptable. If the valuation is for a later version the subsequent growth will not appear so rapid.

For simplicity we consider the initial IP represented by remaining amount of code from Version 1 through the years. In Figure 2 that amount is represented by the darkest blocks. In our example we analyze seven versions, each lasting 18 months, for a 10-year horizon. We adjust the results to the end of each year, since income is counted at the end of each year.

Code scavenging. Some code is scavenged, during maintenance that is, deleted and replaced by new code. In practice the rate of code deletion is small, about 5% code per year, and mainly associated with module replacement. There is little cost to leaving code in place, since the amount of memory and storage available has outstripped the growth of code. But there is a high risk to removal, since some user's program may still depend on features that depend on that code. Removing code takes more time than writing new code.

GW RULE: WITHOUT INCENTIVES, 5% OF THE CODE PER YEAR IS REPLACED DURING MAINTENANCE.

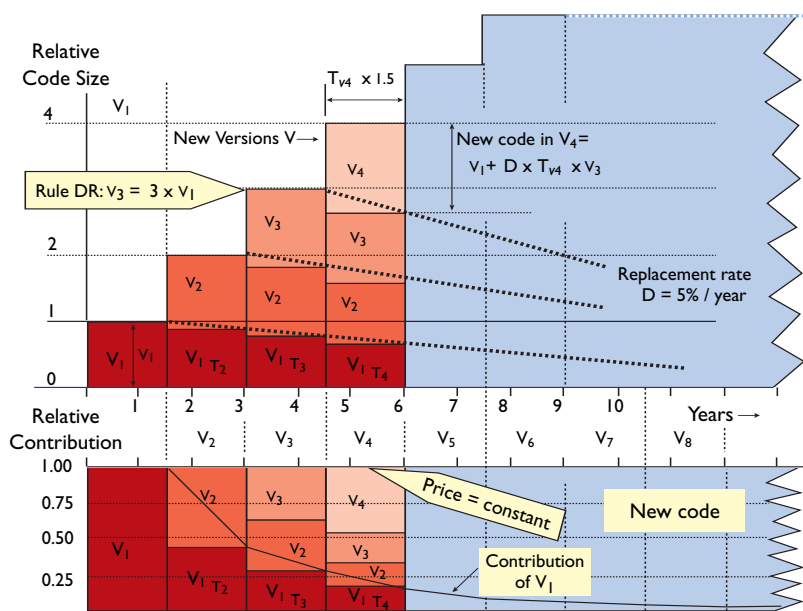


Figure 2. Code growth and diminution of earlier contributions over time.

programming and programming management effort grows exponentially with size, a growth that is more than linear cannot be supported in practice by reasonable growth programming staff. We could validate a rule reported by David Roux that defines a more modest growth rate:

DR RULE: SOFTWARE GROWS AT EACH VERSION EQUAL TO THE SIZE OF THE FIRST WORKING RELEASE.

A similar behavior was observed for operating system modules, as well as its effect on maintenance costs [3].

If we call the first working software Version 1, the DR rule means the expected growth is 100% from Version 1 to Version 2, 50% from Version 2 to Version 3, 33% for Version 3 to Version 4, and so on, as shown in the top of Figure 2. The amount of new

The limit of removal of code for a new version, even after several years, appears to be 20%–25%. Beyond that level it is better to rewrite the code [6]. However, wholly new code entails severe risks of creating incompatibilities for existing customers.

Since the estimate for the total code size in successive versions is independent of the rate of deletion we consider that the deleted amount will also be replaced by new code. Since the code base grows, more code in absolute terms is replaced annually as time goes on.

We also assume that code replacement fractions are equal for existing code and for code added later during maintenance. We can now combine the effect of code growth and code scavenging. If new versions of the software appear every 18 months, then the

amount of new code in Version 2 becomes about 53%, in Version 3 it's 71%, and in Version 4 it's 79%. The influence of the original code and its IP diminishes steadily.

DIMINUTION OF SOFTWARE IP

To relate code sizes to IP value we observe that the price of a software product remains approximately constant, although the code representing that software grows steadily. We use that observation to allocate income.

Income from a software product over time. Although the body of code grows over time, the price of a unit of software sold tends to stay quite stable, typically increasing less than the rate of inflation. From the customer's point of view a new version does not add value, it only provides the scope, reliability, and ease of use that should have been available in the first place. Keeping the price stable for a known and reliable product discourages entry of competition.

P: THE PRICE OF SOFTWARE FOR THE SAME FUNCTIONALITY IS CONSTANT, EVEN IF IT IS IMPROVED.

We now rescale the code contributions based on a constant expected price to obtain the relative contributions of the versions to the price, as shown in the lower part of Figure 2. The rapid growth of software, especially initially, diminishes the value of the initial IP contribution to the product.

Providers of enterprise software commit themselves to deliver any further versions of the software to existing customers, as long as an annual maintenance fee is paid. Such a scheme is attractive to the customer, who can predict expenses into the future, and the vendor, who collects a steady income at low sales costs from efforts that are already required to gain additional customers. Typical rates charged customers for ongoing support are 15% of the original purchase price. Of that amount a small fraction goes into sales support, the effort to get the customer to upgrade to the new version. A larger fraction goes to routine call-center support. The remainder of the support fees, amounting to 6%–10% of the purchase price in every subsequent year, is available to the engineering staff for the types of maintenance presented earlier. That income

also supports most improvements needed to attract new customers to an existing product.

We consider below enterprise software, such as databases and application tools built on top of them. Shrink-wrapped software differs in terms of marketing, but its IP assessment is remarkably similar. For software developed and used in-house benefits are case-specific, and much of next discussion dealing with sales volume, will not apply.

Penetration. If the income per unit of software sold is constant, then income increases are due to the

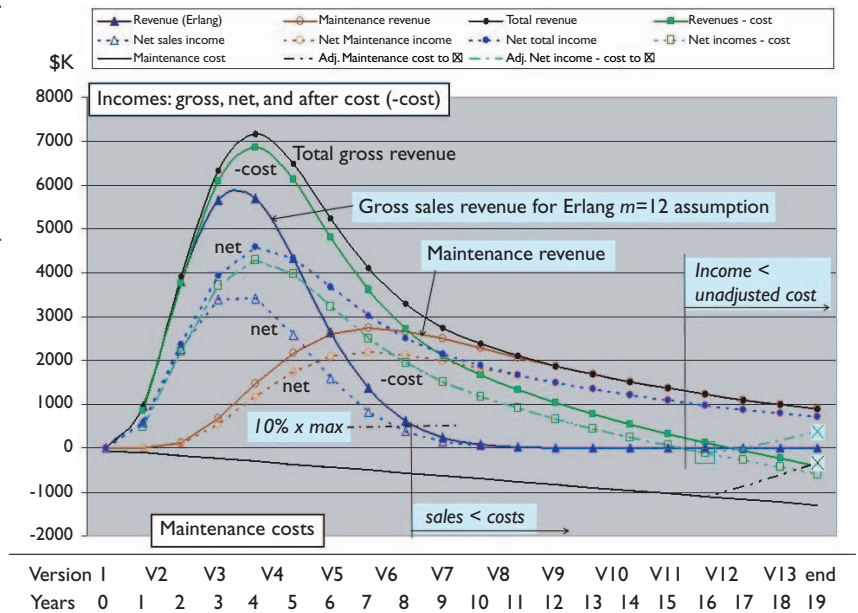


Figure 3. Revenue from sales, maintenance cost, and income, and their combination.

growth of unit sales. Sales can increase until its market is substantially saturated. To reduce uncertainty the size of the candidate market should be estimated using multiple approaches. Common ways include using information about a predecessor product, the number of businesses that need the functionality of the new product and of customers that can afford the product, the number of a certain type of computer or operating system in use, and similar bounds. A 50% penetration is optimistic; beyond that level distortions occur in the market due to the ability to employ monopolistic practices.

A truly useful and well-marketed product can become rapidly known to the customer community via the Internet. A software product can be rapidly delivered to all customers, so that substantial penetration is achievable in a short time. If software installation requires a major effort, then the sales peak will be delayed. Further growth can occur if the number of candidate customers grows.

There is some literature on sales expectations [9]. Simple models use normal curves to describe the

range of customers from early adopters to broad acceptance to laggards. A problem with a normal distribution is that it extends into the past, so the actual point where sales commence is undefined.

A good fit to actual software sales curves has been obtained using Erlang distributions. Just as the normal distribution, an Erlang distribution is also controlled by the mean and variance of the data, but has a definite starting point. Fitting provided data yielded Erlang parameters from $m=6$ to $m=20$. In our sample

Factors	Today	y1	y2	y3	y4	y5	y6	y7	y8	y9
Version	1.0	2.0	3.0	4.0	5.0	6.0	7.0			
Relative size (rule DR)	1.00	1.67	2.33	3.00	3.67	4.33	5.00	5.67	6.33	7.00
Growth R-I	0.00	0.67	1.33	2.00	2.67	3.33	4.00	4.67	5.33	6.00
Replaced (rule GW)	0	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45
Initial code left I-Rep	1.00	0.95	0.90	0.85	0.80	0.75	0.70	0.65	0.60	0.55
Fraction old O/R	100%	57%	39%	28%	22%	17%	14%	11%	9%	8%
Unit price (\$) (rule P)	500	500	500	500	500	500	500	500	500	500
Units sold	0	1911	7569	11306	11395	8644	5291	2740	1241	503
Revenue (K\$) Us×Up	0	956	3,785	5,653	5,698	4,322	2,646	1,370	621	252
SW share @ 25%×Rev	0	239	946	1,413	1,424	1,081	661	343	155	63
From Initial SW F×Ss	0	136	365	400	311	187	93	39	14	5
Discount @ 15% I/I. 15 ^Y	1.00	0.87	0.76	0.66	0.57	0.50	0.43	0.38	0.33	0.28
Discounted share Ss/D	0	208	715	929	814	537	286	129	51	18
IP contribution Fi/D	0	118	276	263	177	93	40	15	5	1
Total disc. share ΣDs	3,687	K\$								
Initial IP value ΣIP	989	K\$								

we use the expected total sales and end when annual sales are less than 10% of the best year. Using $m=12$ yields the gross sales revenue in Figure 3.

Sales based on some fixed IP can never continue forever. Eventually the market penetration is complete, and only replacements are sold. The simple Erlang model shown ignores convoluting factors. For instance, a product that depends on some infrastructure, say Unix, will be affected by the growth or lack of growth of the Unix market. Transborder business is subject to restraints and financial imbalances.

During the rise of the dot-com business boom many projections were faulty. Realistic assessments of consumer markets could have helped. In a freshman class of 1998 we considered the expectations for online toy sales and found that by the year 2003 more toys were expected to be sold on the Web than the total number of toys being sold, an unsustainable situation. Such simple checking of limits and identifying inconsistencies can prevent many problems due to excessive expectations. For instance, infrastructure companies like Cisco and other communication companies that accepted orders from enthusiastic dot-com start-ups, could have realized the total equipment being ordered was greater than the world could absorb and afford.

Expenses attributable to software. For pure software companies gross profit is nearly the same as revenue from sales.

Businesses still have operational costs to be supported by the gross profit: software maintenance, marketing, advertising, sales staff, administration, interest for working capital, and so on. Since no revenue will be generated without such costs these costs reduce the benefits of the initial software. The shareholders will also want their share of the pie, either as dividends or as reinvestments to grow the company.

The required information is available in the financial statements of public companies, as annual reports or published 10-K statements. For a new company one can try to find similar businesses and use their ratios. For our example software company we assign

- 5% to cost of capital—interest on loans, no dividends on initial or venture funds;
- 45% to administrative costs, including sales and distribution
- 25% to software research and development; and
- 25% to marketing and advertising.

Table 1. Summary of software benefit factors over a nine-year horizon.

There can be much variance in these numbers, so data, to be credible, should be aggregated over several years and validated by comparison with similar businesses.

Low capital requirements are typical for the software industry once a company is operational. Low distribution costs are also typical for software, especially if Internet capabilities can be exploited. To gain visibility, however, substantial advertising may be needed.

Our sample company invests in future growth. The investors, being stockholders, expect to benefit in the long term from increased share prices and defer paying taxes. Part of that aspect is captured indirectly in the valuation, namely in the increased value of the product due to ongoing maintenance, as described in the upcoming section on maintenance effect. The software budget is used to improve the current software and to develop new products, generating new IP and hence new value to the stockholders, and is a surrogate for profit.

In this model we take advertising expenses just as a current expense, even though they also increase the IP value of the company. The effects of advertising tend to be short-lived, and have less importance than word-of-mouth recommendations about quality software.

Future allocation. For the valuation, the current

software development fraction investment is continued into the future. It is reasonable to apply the same fraction throughout, unless there is a realistic foundation for higher or lower profit margins in the future. The following example allocates 25% of the net revenue to the software IP component, based on a well-managed company that develops and sells software.

To arrive at today's value of that profit, the benefit of future sales must be discounted. We use here a typical software industry discount rate of 15% per year. In high-risk situations such a rate should be higher.

COMBINING THE INFORMATION

We now have estimates of the net income per unit, the fraction of IP of the original code remaining in each version, and the number of units being sold for a number of years into the future. Those numbers are combined to produce the income associated with the initial software into the future. The benefit of those future sales is then discounted before summing the contributions from each year of the life of the software.

Computing the value of the software. A spreadsheet provides the best means to summarize the evaluation. Table 1 extracts annual values for our example, starting at a Version 1.0 and ending with a Version 7.0. New versions of the software are released every 18 months. For the sake of presentation we limit software life in Table 1 to nine years. The annual IP contribution is that of the first version, the initial software being valued, using the diminution shown in Figure 2.

The result shows that selling 50,000 units of a product at a price of \$500 over nine years yields a bit less than a million dollars in terms of IP value discounted today. For maintenance and growth \$3.687 million were kept available. That amount increases the value of the company, while it also provides jobs and societal benefits.

Note that the collected revenue was much greater than the value of the IP. Getting that income requires spending for ongoing software development, marketing, and all the other items within a business. Figure 3 displays the results extended up to 19 years. Indeed, the sales revenues become negligible after year 9, and because of discounting, the IP beyond that point can be ignored.

We made the simplest possible assumptions to arrive at a clear result. All of the assumptions made

can be challenged and improved in any specific situation. It is also possible to plug multiple assumptions into the computation to arrive at ranges and explore what-if questions. All the values used in the assumptions have pragmatic ranges, and their leverage can be explored. For instance, the results would look much better if the unit price is set to say \$2,000. However, such a change means that fewer customers will buy the software and salesmen will have to work harder.

When we analyze IP, we take the entire income, and then allocate only 25% to the software, since there are substantial other costs. If we want to look at

Factors in services	Today	y1	y2	y3	y4	y5	y6	y7	y8	y9
Est. initial SW cost (\$K)	500									
Cost of maintenance Cm	67	100	167	233	300	367	433	500	567	633
discounted for lag CI	82	123	206	288	370	452	534	617	699	781
SW retained $\Sigma Rev \times 0.9^y$	n.a.	860	4,559	9,756	14,478	17,352	18,262	17,806	16,646	15,233
Revenue left Rev - CI	-82	832	3,579	5,365	5,328	3,870	2,111	753	-79	-529
Maint. revenue $Sr_{-1} \times 0.15$	none	none	129	684	1,463	2,172	2,603	2,739	2,671	2,497
Maint. revenue avail 2/3	none	none	86	456	976	1,448	1,735	1,836	1,781	1,665
Discounted M.rev. Mr/D	none	none	98	450	837	1,080	1,125	1,030	837	710
SW share left @ 25% RI	-21	208	981	1,797	2,307	2,415	2,263	2,015	1,761	1,532
Discounted to today SIxD	-21	181	742	1,182	1,320	1,201	979	758	576	436
9 years disc. total ΣDt	7,350 K\$									

Table 2. Net income including maintenance for the example in Table 1.

net income from sales we consider that some such costs are proportional to sales. Making that cost-of-sales 40% produces the dashed line of net sales income in Figure 3. That amount also becomes negligible by Version 7.

Effect of maintenance. We indicated earlier that maintenance has a high benefit. Here, we investigate that value. The approach represents a service model, where the relationship of supplier-to-customer is maintained after the initial sale. The customer will pay for ongoing maintenance services and always have an up-to-date product. This income is not part of the valuation of the initial software presented previously, since it depends on efforts made in future years.

Two new financial streams come into play now. While there is income from the maintenance fees, the expenses for maintenance must now also be made explicit: *no gain without pain*. Initially we must spend much of the software share shown in Table 1.

Income from maintenance changes the cash flow drastically. After year six the income of maintenance exceeded the income from sales alone. Extending Table 2 beyond year 9 we find that maintenance revenues remain substantial. Figure 3 plots that effect as lines marked with circles. But to generate those revenues maintenance of the product is required, which becomes ever more costly and eventually exceeds the total income, limiting the economic lifetime of the software product for the supplier.

Income from maintenance. A supplier of enter-

The goal of this article is to bring together information from domains that rarely interact directly: software engineering, economics, business practice, and legal sources.

prise software obtains an ongoing benefit by collecting annual maintenance fees. The software earns maintenance revenue every year, even as sales are made to new customers. Most customers renew their contracts since the alternative—obtaining and installing new software—is much more costly than paying a maintenance fee. Table 2 shows the income for 15% annual maintenance fees, based on the initial price of enterprise software. We assume that first-year maintenance is free and that 90% of customers renew their maintenance contract annually. That income lags a great deal behind the sales curve. Not all of the income from maintenance fees is available for software. For instance, there must be a better help desk. For our example, two-thirds of the fee was made available for software.

But maintenance is easier to sell than new software. The dashed line for Net Maintenance income assumes that the annual cost-of-sales to renew a maintenance contract is only 20% of the sales amount.

Cost of maintenance. Now we consider software maintenance costs throughout. Maintenance costs are best estimated based on initial software cost. We assume the initial software cost was \$500K, about half of the computed value obtained in Table 1, leaving the other half to other initial costs. Since we value the workers performing maintenance highly, our model assumes a relatively high 20% annual cost of maintenance, based on aggregated software costs. Maintenance costs do increase over time. A fair way to estimate that growth is to make the cost proportional to the total, ever growing code size, as shown in Figure 3. The actual maintenance cost from year 2 to year 7 now amounts to \$2,075K. We find that long-term maintenance consumes about 80% of the total cost, a typical percentage.

The maintenance costs are borne by the income from ongoing software sales and maintenance licenses. Those costs overlap the costs needed to keep the software product viable for sales. Since the benefit of maintenance expense is delayed, but paid for with current funds, it must be discounted—we must compute a lag; that is, the delay between the time of the investment and the time that the investment

brings in income. Given a development time of one year for maintenance code, testing for six months, and new versions every 18 months the average lag is assumed here to be at 1.5 years. That means that effective maintenance costs are increased by a steady 23% to account for the 15% annual discount. Still, by the third year the allocated income from maintenance fees is sufficient to pay for all maintenance.

The top curve in Figure 3 shows the total revenue from sales and maintenance, and shows that revenues are still substantial after 19 years, when all are due to maintenance fees. But maintenance costs increase, and by year 17 the operation shows a loss. The loss is more pronounced if we consider the net incomes, shown as dashed lines. Now we have a loss in year 16, indicated by the large square in Figure 3.

It is wise to stop improving the product at that time. Stopping abruptly generates bad press for a company that sells other products as well.

Gradually reducing expenses as less income is generated is wiser. Still, when customers find out that maintenance is reduced, sales will diminish rapidly and fewer maintenance licenses will be renewed. We also show in Figure 3, as the dashed lines terminating in a boxed cross, what happens when we reduce maintenance costs 25% per year and suffer a corresponding reduction in maintenance contract renewals.

Even with the high cost of long-term ongoing maintenance the actual cash flow increases nearly three times over the nine-year timeframe. However, since these additional benefits occur in later years, the income discounted to today increases less. That income is about twice as much as the income without maintenance revenues.

Model differences. Obtaining a result for the value of maintenance required many more assumptions than the base IP valuation for the initial value of the software. We had to consider ongoing costs as well as income. To obtain a base for maintenance costs we also had to guess the initial cost.

The beneficial difference of including maintenance efforts is not captured in a traditional IP valuation. Operating in an effective service model also requires changes to business practices. Modeling an enterprise

business with maintenance obeys the appraisal rule that a business should be valued as it actually operates.

CONCLUSION

Valuation of software is not easy, but is feasible. The novel contribution of this work is the software growth model, but to assess its effects we must place it in an overall business model that allows a quantitative assessment of the value of the inherent IP. Without such a model unrealistic amortization schedules tend to be used. Many assumptions are required to assess the software-related income and costs, but making assumptions is inherent in dealing with the future. The alternative—remaining ignorant—provides no guidance. Evaluating alternate assumptions with a spreadsheet based on the model provides yet more insights. For instance, determining the benefits of maintenance as a service is difficult if we can't even tell what the value of software is in the first place.

A specific conclusion is that maintaining software, although costly, is very worthwhile. Maintenance provides continuing refreshment of the inherent IP. With high-quality maintenance a company can transform its business from a sales model into a service model. Service income is attractive to mature companies, reducing the need for new products to keep sales high. To operate in a service model management must appreciate the process and motivate software staff to sustain the value of the product. Everyone must be involved, not just the programmers, but also the staff that provides the ongoing intellectual input for the three types of maintenance presented in this article.

However, maintenance costs eventually overtake income. The limit on the life of software is not obsolescence, but rather the cost of maintaining ever more complex software [3]. Providing for ease of maintenance in software design and managing maintenance well should have significant benefits.

Education and technological attitudes favor novelty over maintenance. Typically software engineering textbooks devote less than 1% of their pages to maintainability and maintenance, although they admit that maintenance costs are greater than 60% of total cost.

Most students in computing disciplines graduate without ever having faced the issue that software must adapt. Students might have had a summer job in a company that assigned them to maintenance tasks, because the knowledgeable programmers wanted to move on and do new stuff. It is, of course, an illusion that cheap labor reduces the cost, it mainly reduces the benefits of maintenance. Managers bemoan the high cost of maintenance, since they are not clear about the benefits [12]. It would be better to let

interns and recent graduates do innovative work, while rewarding the experienced staff who sustain the value of their existing products and services in other ways, leading to a “gentrification” of maintenance.

The analysis approach shown here cannot deal with the valuation of open source software, which also requires maintenance [12]. Companies that tried to create a business from servicing open source software have had a very difficult time, since reasonable maintenance costs seem extremely high when acquisition is nearly free.

Even though the analyses to value software depend on assumptions that cannot be verified in advance, having the assumptions stated explicitly and analyzed allows a discussion of alternatives. Opinions and disagreements are brought down to a less emotional level, and will include more organizational and technical specifics. **■**

Contributing to this work were Treasury economists, Charles Adelberg, Joy Yen, and Shirley Tessler. Any errors in this article are, of course, my responsibility, but I will not assume any responsibility for business decisions based on application of the presented model.

REFERENCES

A broader version of this article with additional references, software business types, and downloadable spreadsheets that can be used to experiment with the models is available at infolab.stanford.edu/pub/gio/inprogress.html#worth.

1. Abba, W. Earned value management: Reconciling government and commercial practices. *Program Manager* 26, 58–69.
2. Basili, V. Viewing maintenance as reuse-oriented software development. *IEEE Software* 7, 1 (Jan. 1990), 19–25.
3. Belady, L. and Lehman, M.M. An introduction to growth dynamics. *Statistical Computer Performance Evaluation*. W. Freiberger, Ed. Academic Press, 1972.
4. Boehm, B. *Software Engineering Economics*. Prentice-Hall, 1981.
5. Damodaran, A. *The Dark Side of Valuation: Valuing Old Tech, New Tech, and New Economy Companies*. Prentice-Hall, 2002.
6. Glass, R.L. *Facts and Fallacies of Software Engineering*. Addison Wesley, Reading, PA, 2003.
7. Jones, T.C. *Estimating Software Costs*. McGraw-Hill, 1998.
8. Lev, B. *Intangibles, Management, Measurement and Reporting*. Brookings Institution Press, 2001.
9. Mahajan, V., Muller, E., and Wind, Y., Eds. *New-Product Diffusion Models. International Series in Quantitative Marketing*. Kluwer, 2000.
10. Pigoski, T.M. *Practical Software Maintenance—Best Practices for Managing Your Software Investment*. IEEE Computer Society Press, 1997.
11. Smith, G. and Parr, R. *Valuation of Intellectual Property and Intangible Assets, 3rd Edition*. Wiley, 2000.
12. Spolsky, J. *Joel on Software*. Apress, 2004.

GIO WIEDERHOLD (infolab.stanford.edu/people/gio.html) is an emeritus professor of computer science, electrical engineering, and medicine at Stanford University. He is a fellow of the ACM, the IEEE, and the ACMI.

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.