

What are Web Services Worth?

Gio Wiederhold
28 January 2005

Abstract

This work presents a method for valuing software available on the web. The semantic web is based on the assumption that there will be a <gaggle> of providers of services, from which consumers can pick and chose. Many technical issues are being addressed, but less study has been devoted to the economic issues. Without measurable benefits for the suppliers of web services the growth of the semantic web is likely to be slow, even as technical issues are resolved.

Economically, the value of software is based on the income that use of that software is expected to generate in the future. That income can be direct, in terms of payments for services, or indirect, as a cost of selling other services and products, as travel or electronics.

The approach described applies well known principles of intellectual property (IP) valuation, sales expectations, discounting to present value, and the like, always focusing on software services, including the costs and benefits of ongoing maintenance. All steps of the process are presented and then integrated via a simple quantitative example. Having a quantitative model on a spreadsheet allows exploration of a service business model. Some conclusions are drawn that reflect on academic and business practice.

1. Introduction.

There exists a voluminous literature on estimation of the cost of producing software, but that literature largely ignores the benefits of using that software [Boehm:00]. Software creators were rarely called upon to quantify its benefits, and it was left to lawyers, economists, software vendors, or promoters to assign value to software products [Stobbs:00] [SmithP:00] [Lipschutz:04] [Bonasia:02]. The results are often inconsistent. Now the move to web services brings many developers closer to the customers. With little history to provide guidance, it is hard to set prices for services or charges for sales. If prices are set too low, then quality services cannot be sustained. Raising prices later is awkward and discourages use. Setting prices initially too high discourages adoption of web-based solutions and broad incorporation of the services.

This work does not deal with how income should be collected: by charging transaction fees, subscriptions, or relying holly on indirect income. However, understanding how much income is needed will help in making those tradeoffs.

1.1 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 value of houses they design, a potter will know the price for the dishes to be sold, as will a builder of bicycles. But software is easy to replicate at a negligible cost. That does not mean that those copies have no value. Providing software as a service adds greatly to the cost and increases the business risk of liberal distribution of the software. To estimate future income requires estimating the potential service sales volume. Predicting the quantity of any kind sales is hard, and there are few experts available to help.

The value of a book, of software, and of software services is essentially 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 wants have little value. Awareness of the value of the product of ones knowledge and effort can help in making decisions on the design and the degree of effort to be spent in creating and maintaining the service. The motivation leading to the generation of this article is to increase the awareness by members in the computing community how the result of their work may be valued. That should, in turn, affect the practice of software engineering and software services.

1.2 Protection

There is substantial literature on the protection of the intellectual property value associated with software. Providing services instead of selling software products greatly increases the protection possible. Internal mechanisms can remain a trade secret. However, once a service is visible on the web, its functions can be reengineered by competitors. Copyright and patents afford little protection. The provision of quality data, up-to-date interfaces, and service quality is likely more important to success than legal protections.

1.3 Outline <review>

In the remainder of this article I will first present the principles of valuing intellectual property, with a narrow focus on the income generated by a software product over its lifetime. The valuation itself addresses software as it exists at some point in time, and ignores the costs of its creation. Once the value of the product is known or estimated, one can compare that value with the cost of its creation and decide if the project is profitable, or, if it was not, what must be changed to make it so.

Software, since it grows over time, presents novel issues, not seen in other intangibles, as music and books. Maintenance to sustain software effectiveness occurs throughout the time that the software is in use, i.e., while the software actually generates benefits. Over time maintenance costs typically exceed the original software development cost, often by factors of 3 to 10. Maintenance causes growth of software, and Section 3 will show how three types of maintenance efforts together affect growth and value. In Section 4 the growth of software will be modeled, using some rules-of-thumb, i.e., rules derived from experience. In order to quantify values we need some metrics. All of software engineering suffers from inadequate metrics, and valuation is no exception.

In Section 5 we predict sales volumes for sold software. Fairly standard business methods are used, selected for their suitability for analyzing software sales. Finally, in

Section 6 we combine results from the prior chapters to arrive at the actual assessment of the value of software. A sample computation integrates the various topics. To illustrate the use of a quantitative model for analyzing alternatives, we show a model where maintenance income is included, representing a service-oriented business model. The conclusion provides some advice, for individuals, managers, and even educators.

This article brings together information from domains that rarely interact directly: software engineering, economics, business practice, and legal sources. We use a number of rules <of experience>, and set them off. The references include some citations to each of the contributing domains, but those cannot begin to cover more than top-level concepts. There are hence relatively many references to books, and fewer to recent topical articles, although many articles have helped to provide the insights presented here.

2. Principles of IP Valuation

Assigning value to intangible property and services is assuming greater and greater importance, as our society moves from dependence of hard, tangible goods to a world where knowledge and talent creates the intangible goods we desire and need. In the tangible world goods are produced by a combination of labor, capital, machines, and management, but the quality of the human components plays a minor role in valuing a company. Even today, the book value of a company shown in its annual report is merely the sum of its facilities, inventory, equipment, and finances. This sum has little to do with how investors value companies in the software domain [Rechtman:01]. For example, we can look at Salesforce.com's Quarterly Report for October 2004 and find out that its book value (assets - money owed) was about \$247M. But its shareholders valued the company at \$1,390M, using the market price of the shares (\$13.58) and the number of shares in circulation, about 102M. The investors in Salesforce.com base the market value on the income they expect to obtain over time from their shares [Becker:02]. The difference, \$1,143M, is due to the intangible property (IP) owned by Salesforce.com and its shareholders. No report breaks down that intangible value to the same detail that the book value is documented.

2.1 The value of IP

Investors in a software enterprise assert through their purchase that

| |
|--|
| IP rule: <i>The value of the Intellectual Property is the income it generates over time</i> |
|--|

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 [SmithP:00].

The IP owned by Salesforce.com, or any company relying on intangibles, 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. Valuation of all that IP is required when companies or business lines are purchased, and many approaches compete [Damodaran:02].

This article focuses only on software and services provided by software. Software is actually the most tangible of the intangibles owned by software service companies.

2.2 Estimating income

To value the IP inherent in software, one must estimate how much income the software will bring in during its future life, which in turn requires estimating its life. We can distinguish six types of software businesses

1. Companies that build software for internal use
2. Companies that use software to provide external services to help sell their goods, as airlines
3. Companies that sell services based on software, as information vendors
4. Companies that produce enterprise software to be installed at customer sites, for sale and long-term lease, as large database systems, financial and business systems
5. Open-source organizations, that give software away, but expect income from ongoing maintenance
6. Companies that produce software packages for sale, as word-processing programs and spreadsheets.

Rather than focusing on their differences, this analysis considers their commonalities. Similar software can produce income according to several of these six categories. Where they differ in terms of income streams, parameters in analysis scheme can take care of those distinctions.

For instance, software that provides inventory information, initially used internally, can be made available to help customers plan their purchases. Aggregated information, say sales and supplies of product categories over time and by area, may be sold to customers doing strategic planning. If the customer needs a better response time, that software may be installed on customer site, but will require maintenance. Major software may be initially sold, but a maintenance contract is needed to keep the software functioning adequately over time. Periodically new version will be issued. When packages are being sold without substantial maintenance, they will become obsolete in several years. But a vendor will have developed successor versions, and most customer will purchase those eventually, or if some versions may be skipped.

What is the technical commonality? In each case, there was an initial investment leading to a product, and if that product is successful, ongoing maintenance is performed in order to maintain a business. With responsive maintenance software can easily live 10-15 years and hence continue to contribute to long-term generation of revenue.

In the remainder of this paper we concentrate on companies that provide software services, categories 2 and 3 above. Other categories are analyzed in an earlier paper, although the principles are identical [Wiederhold:05].

2.3 Net revenue

In business evaluations, as annual reports, the revenue realized is immediately reduced to net revenue by the cost of the goods sold. Here is where software and much other intellectual property differ. The effort to make the first unit of software is the overwhelming cost, accounted for as research and development. For software the actual manufacturing cost is negligible, it may only require copying CDs or putting the code on a website. Tangible goods incur a substantial manufacturing cost for each successive unit.

To assess the value of existing software, we ignore now its initial research and development cost. Then, once the value is known, we can see if its creation can generate profits. When providing services, the income per unit is equal to the product of the prices charged per use and the usage volume.

3. Sustaining Software

Before we can proceed moving from the income generated by software to valuation of its IP we must consider what happens to software over the time that it generates income. It is here where software differs crucially from other intangible goods. Books and music recordings remain invariant during their life, but service software is rarely stable.

Methods used to depreciate tangibles as well as intangibles over fixed lifetimes are based on the assumption that the goods being valued lose value over time. Such depreciation schedules are based on wear, or 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.

All substantial business software must be sustained through ongoing maintenance to remain functional. What maintenance provides was stated many years ago by Barry Boehm [Boehm:81, p.533]:

".. 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"

Ongoing maintenance generates IP beyond the initial IP, and will have to be considered in the valuation.

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 10 to 15 years, and are likely to increase [SmithP:00] [Wiederhold:95]. Version frequency is determined by the rate of changes needed and the tolerance of users to dealing with upgrades. In our example we will assume a steady rate of 18 months, although new software versions may be issued more frequently, while the rate reduces later in its life.

3.1 Continuing improvement

We use well-established definitions for the three classes of long-term maintenance [Marciniak:94]. Other, more detailed lists of maintenance tasks have been provided [Jones:98], but those tasks can be grouped into the three categories below, which distinguish the tasks by motivation, timing and feedback mechanisms [IEEE:98]. Feedback issues, crucial to IP generation, are detailed in Section 3.2

1. **Bug fixing or corrective maintenance** is essential to keep customers. In practice, most bug fixing is performed early in the post-delivery cycle – if it is not successfully performed, the product will not be accepted in the market place and hence not have any significant life. There is substantial literature on the benefits of having high quality software to enable reuse, a form of long life, but those analyses document again cost avoidance rather than income [Lim:98].

2. **Adaptive maintenance** is needed to satisfy externally mandated constraints. Such changes allow the software to deal with new hardware, operating systems, network, and browser updates that are used in the customers' environment. Governmental regulations may also require adaptations, new taxation rules affect financial programs, accounting standards are upgraded periodically, etc. All such changes must be complied with if the software is to remain useful. Within a business new mergers and acquisitions force changes in information systems [Pfleeger:01], and new medical technologies affect health care software [BondS:01].
3. **Perfective maintenance** includes performance upgrades, assuring scalability as demands grow, keeping interfaces smooth and consistent with industry developments, and being able to fully exploit features of interoperating software by other vendors, as databases, complementary webservices, schedulers, and the like. Perfecting makes existing software work better. In that process the functionality is upgraded, but not to the extent that truly new products are created. Perfection may be less urgent, but keeps the customer happy and loyal [Basili:90].

Maintenance costs of enterprise software amount to 60 to 90% of total costs [Pigoski:97]. Bug fixing, for software that is accepted in the market, eventually reduces to less than 10% of the maintenance effort [LientzS:80]. Adaptation consumes 30% to 40% of the maintenance costs, and varies with the number of interfaces that have to be maintained. Ongoing perfection is known to require 45% to 55% of maintenance costs over the long term. The effectiveness of maintenance is greatly reduced by poor design and lack of adequate documentation [BeladyL:72]. Figure 1 provides a simple depiction of the distribution, in practice it will differ depending on the setting and on external events.

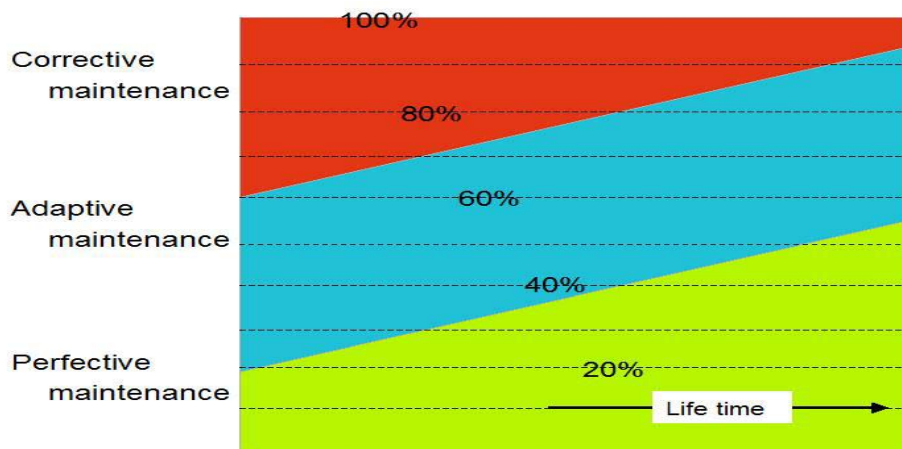


Figure 1: Maintenance Effort over the Lifetime of software

There are two aspects that motivate a service provider to provide a high level of maintenance: one is to keep the customer -- it is easier to keep an old customer than to gain a new one -- and the other is income -- good maintenance has a high value to the customer and can generate substantial and stable income [Wiederhold:03].

3.2 Sources of IP in the Maintenance Phase

Maintenance is often viewed as work requiring little expertise. That view is only true to the extent that the information sources for maintenance are diverse.

Corrective maintenance is based on a feedback from the users and customers. A specialized team at the provider will log error reports, filter stupidities, eliminate misunderstandings, combine faults that seem to be identical, devise workarounds if feasible, advise the users, and forward any remaining significant problems to the engineering staff. If the original developers are still involved, fixing bugs becomes an opportunity to learn, rather than a task to be dreaded. Corrections are often needed when special cases or novel combinations are identified. Code to deal with those cases is added, and the software grows.

Adaptive maintenance is required to sustain services so they keep up with external changes of its settings. Input for required adaptations will come from providers who market services that should mesh with ones product, from standards bodies, from hardware vendors, and from government sources. Major providers will have representatives at external organizations and vendors. Filtering and scheduling this work requires a high level of skill. It is rare that prior interfaces and capabilities are removed, the software grows.

Perfecting is also an ongoing process. Any attempt to write perfect software will induce excessive delays. Perfection in software is a process, not its result. Perfecting does not require novel insights, just an ability to listen and act. There is no science that 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 should be catered to. All perfecting changes will be based on feedback from ongoing use, but should not disturb current users [Aron:83]. Code that considers the expectations of the users will be larger than code is organized for the convenience of the programmer. The process of perfecting software is a major component of growth.

4. Growth of Software

Since software IP is embedded in the code being used, and that body of code changes over time, we must now evaluate what happens to the code and its functionality. For convenience we use lines-of-code (LoC) as a metric of size. That metric will be available at the service provider, although LoC certainly has its limitations [Wiederhold:05].

4.1 Growth of code

The maintenance activities that sustain software cause the software to grow in size, as presented in Section 3. Hennessy and Patterson, coming from a hardware view, have presented a rule [HennessyP:90]:

HP rule 5: *Software, in terms of lines-of-code, grows by a factor 1.5 to 2 every year.*

However, this rule implied exponential growth, expected for hardware, but such growth cannot actually 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.*

A larger growth creates too many problems and will not allow the newly released version to be reliable [Bernstein:03]. The existence of a limit is clear from Fred Brook's essays: since programming and programming management effort grows exponentially with size, an exponential growth of software cannot be supported in practice by reasonable growth programming staff [Brooks:95]. Cost estimation tables further support such barriers [Jones:98]. We were able to validate a rule, which defined a more modest growth in enterprise settings [Roux:97]:

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 [BeladyL:72]. There is no obvious reason why it should not apply in web services as well.

If we call the first working software Version 1 -- not always done in practice -- the DR rule means that 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, etc. as shown in Figure 2. The amount of new code due to growth is 50% in Version 2, 33% in Version 3, 25% in Version 4. By that time growth of code obeys Bernstein's rule (PB) 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 limit initially seems acceptable. If the valuation starts with a later version, i.e., more mature code, then the growth will not appear to be so rapid.

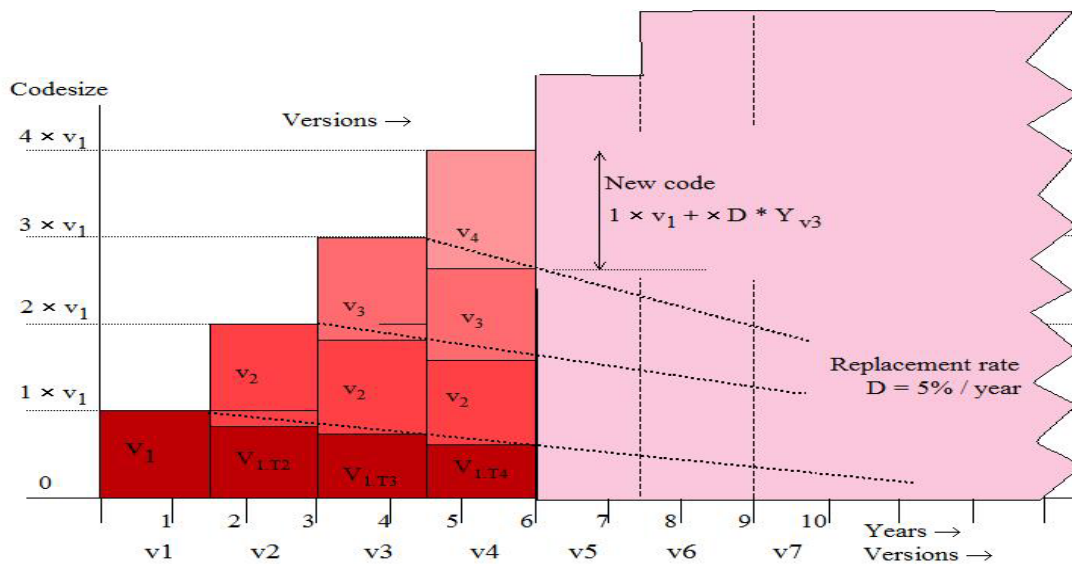


Figure 2. Code growth and reduction of earlier contributions over time.

For simplicity we consider the IP represented by the code remaining from Version 1 through the years. That code is used as a surrogate for the original IP. In Figure 2 that amount is represented by the bottom, darkest blocks. In our example we will have 7 versions, each lasting 18 months, in the 9-year horizon. We adjust the results to the end of each year in Table 1, since income is counted at the end of each year.

4.2 Code Scavenging

During maintenance some code is deleted or scavenged, i.e. deleted and replaced by new code. We found that the rate of code deletion is small, and mainly associated with module replacement. Such behavior is not surprising. The amount of memory and storage available has grown faster over time (exponentially) than the code. There is hence little cost to leaving code in place, even if it is effectively dead. But there is a high risk to removal, since some usage may still depend on undocumented features that depend on that code. Removing code takes more time than writing new code. From code inspection we see about 5% code removal per year, going up to 10% for code that is aggressively maintained.

GW rule: *Without incentives, 5% of the code per year is replaced during maintenance.*

Since the code base grows, more code in absolute terms is replaced annually as time goes on. Since the estimate for the total code size is independent of deletion we can consider the amount being replaced to also to represent new code.

For simplicity we also assume that code replacement fractions are equal for old code and for later code that has been added during maintenance. We can now combine the effect of code growth and code scavenging. If new version 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. Growth also means that maintenance costs grow, but we consider all new IP to be represented by new code, and focus now only on IP retention represented by old code.

4.3 Relative Value of Old and New Code

We presented growth of code to get a handle on valuing its IP. A reasonable assumption, 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.

Code that was added during maintenance has value in terms of providing smooth operation and a high degree of reliability. The original code provided the functionality that motivated the customer's usage in the first place. If that functionality were inadequate the customer would have found alternatives and be unlikely to come back. However, new code may include some new adaptations or perfections that motivate some additional usages. Given that the positives and negatives can balance each other out, it is reasonable to assign the same value to lines of old and of new code.

5. Devaluation of Software IP

To relate code sizes to IP value we use the earlier observation that the price of the software service remains constant, although the body of the code representing that software has grown steadily. We can then allocate that income over time.

5.1 Income from software services over time

There is today little data on the pricing of software services. Although the body of the code delivering services will grow over time, the price of a unit of software service is likely to stay quite stable. Even increasing prices at the rate of inflation may be hard. There are several reasons here. From the customer's point of view a new version does not

add functionality, it only provides capabilities that should have been available in the first place.

P: The price of the same functionality is constant, even if its scope improved.

Raising prices for services in the public view and functionally well understood provides an incentive for customers to look around and for new competitors to gain a foothold. For software services that support sales of corporate products it is harder to discern clear strategies. Internal pricing is rarely well documented, even when internal cost-centers are established. Some data can become available when such services are outsourced.

We can now rescale Figure 2 based on a constant expected price v_1 , and see how the rapid growth of software, especially initially, reduces the value of the initial product. The result is shown in Figure 3. One can argue that the first version contained all the truly original IP, so that the diminution should be less steep. However, to realize that value over time ongoing work is required.

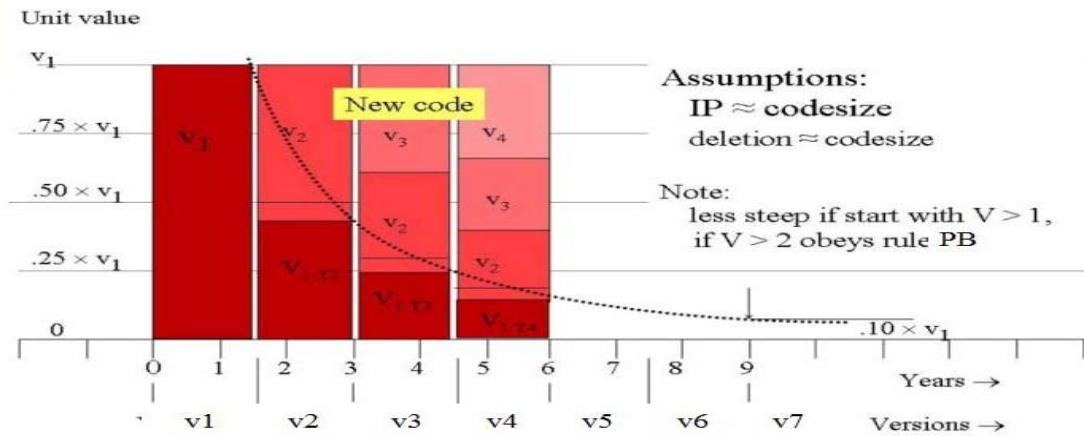


Figure 3. Diminution of the original IP due to maintenance.

Suppliers of web services will steadily improve their services to gain additional customers. These ongoing improvements are also made available existing customers. Such a scheme is attractive to the customers, who can predict expenses for their usage into the future, and the vendor, who collects a steady income at low sales costs and no incremental development costs.

5.2 Penetration

If the income per unit of software service is constant, then income increases are mainly due to increases in the quantity direct or indirect sales. The use of software can increase until the market is substantially penetrated. For an established market the size of the candidate market can be estimated based on data about a predecessor product's sales, the number of businesses that need the functionality, and the like. Web services have little track record, so that much growth can be expected over time. Yet, there will be limits.

To estimate eventual sales one must determine how the function provided on-line is provided today, what the number of customers and their usage might be, and, if product sales are involved, what the fraction of products are likely to be sold on-line. We made some estimates of the on-line toy market and its projections in 1999. The projections

showed that 115% of all toys sold would be purchased on-line by 2002 [WiederholdC:98]. We all know what happened to those companies before they reached that year.

Estimating the future is always hard, but there are boundaries. Absolute upper limits can be based on fractions of gross economic data values as population, logistical limits, and even gross national products. For instance, the sales booked by communication infrastructure companies in the heady dot.com years implied that spending in the U.S. on communication would increase by an order of magnitude. In any case, a 50% penetration of existing markets is optimistic; beyond that level distortions occur in the market due to the ability to employ monopolistic practices, which I'd rather ignore. Economists also model diffusion rates for new products, but that approach implies that there is an infinite market, without saturation [MahajanMW:00]. The ease of distribution of software means that a useful product can be rapidly marketed to all suitable customers, so that sales ceilings are reactively near-term. To sell more, true product innovation is needed, but that issue is not part of our discussion here.

A good fit to known software sales curves has been obtained using Erlang distributions [ChatfieldG:73]. That distribution is also controlled by the mean and variance of the data, but has a definite starting point when sales start. Computing the best matches to data yielded Erlang parameters from $m=6$ to $m=20$, but clustered at $m=11$ [Wiederhold:83]. At that value the distribution appears similar to the normal curves of Moore [Moore:95], but is foreshortened and has a maximum rate before its midpoint. We show in Figure 4 such a curve for sales of about 50M service requests over a relatively short life, and also curves corresponding to two other Erlang parameters. The areas under each curve represent total sales and should be equal up to year 9. At values of m close to 1 an Erlang distribution resembles a negative exponential - corresponding to ever-decreasing sales over time; at $m=\infty$ it is a constant, corresponding to a one-time special sales promotion. Erlang curves have been widely used to size communication channels, and a variety of software is available to compute m for known mean and variances. For software it's best to start with the expected total sales and a sales horizon ending when annual sales are less than 10% of the best prior year, as done here for $m = 12$.

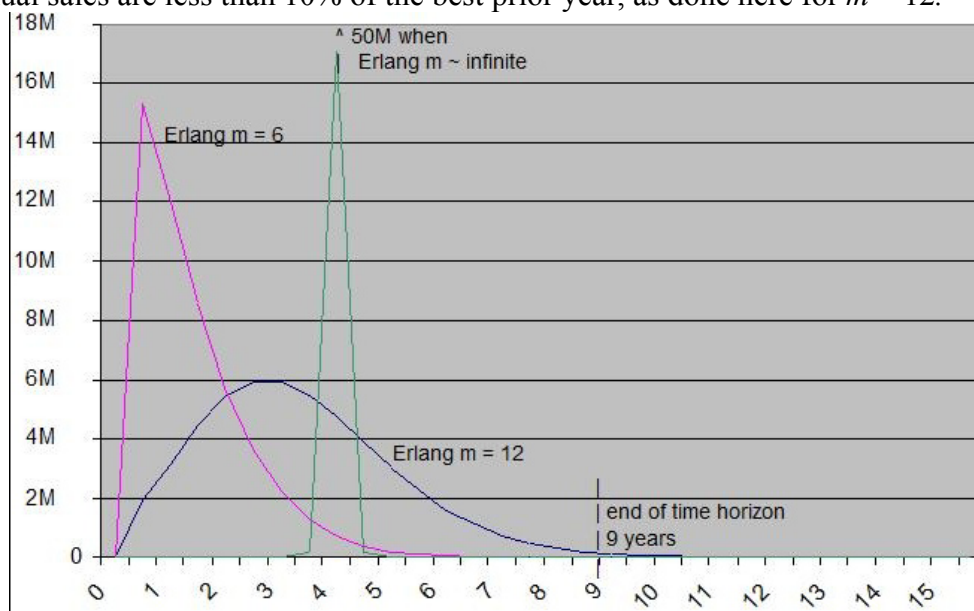


Figure 4. Semi-annual sales for an expected 50M services over 9 years, and continuing.

Some routine web services, part of routine, ongoing business activities, might have longer, flatter sales curves. Reducing the variance will generate such distributions. At this time, we don't have sufficient data to model such sales curves. In general, sales based on some fixed IP, even if maintained, can never continue forever. Eventually, these services become commodity services, and prices will become an issue. In our model, we kept prices fixed, and reduce sales volume.

5.3 Income Attributable to Software

Sales provide the revenue needed for software businesses. But businesses have operational costs that are not directly related to sales: administration, sales staff, interest for working capital, etc. And on a company's net income taxes will have to be paid. Eventually the shareholder's will want their share of the pie, either as dividends or as reinvestments to grow the company.

In a service company we can consider three types of income

1. Routine or commodity income. This fraction represents the income that would be received without any IP contribution. For instance, a public information service at a government agency, relying only on internal funding, and not trying to compete, can be considered as having only a commodity income equal to its funding. A complicating factor is the value of prestige and political goodwill in such a situation. In many traditional businesses, where customers arrive just because the business exists, routine income can be a large fraction of the total income.
2. Income generated by technical innovation and the resulting IP. In our case software would be the prime component, but any items that are the result of research and development expenses should be considered, as new mathematical methods to present results in a more relevant form. Acquired software may contribute to the innovative aspects, or may simply support routine operations.
3. Marketing and advertising expenses are justified by increasing sales over what routine and research efforts can generate. These costs are considered as also contributing to IP, the results add to the value of a company without appearing in the corporate books. Marketing costs include expenses that actually motivate the company and its employees to make quality products, as well as to the public perception of quality.

To allocate income to software the relative magnitude of these types must be estimated. If there are companies providing routine services, and their relative income can be used as a basis, and any excess income then allocated to the IP generating efforts (IGEs). If there are other companies doing a similar business, and their IGE expenses are known then a comparative routine income can be established there. A bottom fraction would be the cost of capital for the enterprise to get started and operate, because a business that routinely earns less than the cost of capital needed to sustain it is not viable.

The IGE costs can be allocated among software and marketing. But since software typically has a much longer life than advertising expenses, the aggregation over time has to be considered. If there is a long and steady history, then the allocation can be based on past expense ratios.

Specifics on how companies allocate their income can be obtained from annual reports or published 10-K statements. For our example software service company we assigned 25% to software development, aggregating several years of research and development costs. Note that we have been focusing on income, not profit. Profit is the excess of income over all costs. Also, profit is taxable. For the example the only taxable component was payment of dividends, included in the cost of capital.

Most startup companies, instead of paying dividends, invests in growth, spending all of their excess income on software development and advertising. The stockholders expect to profit from increased share prices. That aspect is captured indirectly in the valuation, namely the increased value of the product due to ongoing maintenance. If the software budget is also used to develop new products, that generates new intellectual property and hence new value to the stockholders, and is a surrogate for profit.

5.4 Discounting

To arrive at today's value of that profit the benefit of future sales has to be discounted into the future. We use here a pragmatic discount rate of 15% per year. In high risk situations such a rate should be higher. Corrections, using appropriate values of beta can be applied [Barra:98]. Finding the right beta values requires matching the software company to similar businesses and their experience. Here we just assume the sample company to be average risk and use a beta of 1.0.

6. 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 can be combined to produce the income associated with the software into the future. The benefit of those future sales is then discounted to today. Finally we can sum up the contributions from each future year in the horizon.

We now need to make some estimates of costs. We assume a modest initial cost of developing the software of \$1.0 million. That number can later be validated for reasonableness, but is essentially arbitrary, and other values can easily be inserted in the spreadsheet. Software maintenance, for well-designed software in a novel setting, tends to cost about 20% of the initial cost, but will increase with the growth of the software. We assume the growth in cost to be proportional to the size, again an assumption that holds when software is modular, so that maintenance does not become excessively complex as it grows [Jones:98].

6.1 Computing the Gross Income

A spreadsheet provides the best means for the evaluation [Wiederhold:05S]. Table 1 extracts annual values for our example, starting at a Version 1.0 and ending with a Version 7.0. The simple general formulas used by economists, especially if they assume perpetual benefits, do not allow for the variety of factors we consider here [Gordon:82]. Since we model only software costs, and not the other costs of doing business, we define a gross income to be the revenue from the services minus the software maintenance. That maintenance cost is equivalent to the cost-of-goods-sold (COGS), the term used in business models for the cost of tangible goods. Current accounting rules will have a problem with this equivalence.

| Factor | Today | Y 1 | Y 2 | Y 3 | Y 4 | Y 5 | Y 6 | Y 7 | Y 8 | Y 9 |
|---------------------|--------|------|-------|-------|-------|-------|-------|-------|--------|--------|
| Version | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 | 6.0 | 7.0 | | | |
| Initial SW cost k\$ | 1,000 | | | | | | | | | |
| Service price \$ | 0.50 | 0.50 | 0.50 | 0.50 | 0.50 | 0.50 | 0.50 | 0.50 | 0.50 | 0.50 |
| Relative size | 1.00 | 1.67 | 2.33 | 3.00 | 3.67 | 4.33 | 5.00 | 5.67 | 6.33 | 7.00 |
| Annual services k | 0 | 1911 | 7569 | 11306 | 11395 | 8644 | 5291 | 2740 | 1241 | 503 |
| Revenue k\$ | 0 | 955 | 3,785 | 5,653 | 5,698 | 4,322 | 2,646 | 1,370 | 621 | 252 |
| Maintenance@20% | 0 | 200 | 333 | 467 | 600 | 733 | 867 | 750 | 378 | 211 |
| Gross income k\$ | 0 | 755 | 3,451 | 5,186 | 5,097 | 3,589 | 1,779 | 620 | 243 | 40 |
| M/SW ratio | | 1.06 | 0.39 | 0.36 | 0.47 | 0.82 | 1.95 | 4.84 | 6.23 | 20.8 |
| Discount% @15% | 100 | 87 | 76 | 66 | 57 | 50 | 43 | 38 | 33 | 28 |
| Disc.d gross inc. | 0 | 630 | 2,397 | 3,001 | 2,458 | 1,442 | 596 | 173 | 56 | 8 |
| Gross inc. NPV k\$ | 13,038 | | | | | | | | Actual | 50,599 |
| SW maint. cost k\$ | | | | | | | | | Actual | 4,539 |
| SW share @25% k\$ | | | | | | | | | Actual | 5,190 |

Table 1. Summary of software service benefit factors over a 9-year horizon.

We have made the simplest possible assumptions to arrive at a clear result. Selling 50M services and maintaining those services at a \$0.50 price per service over 9 years yielded about 12.5 million dollars gross income when discounted to its net present value (NPV). However, the ratio of actual maintenance costs to the 25% allocated for software (M/SW) showed a deficit (1.95) in year 6. To avoid a loss in successive years, management would decide to reduce perfective maintenance and to close the service down in year 9. In practice, sales of services will diminish once the customers find out about our intent, but this is ignored in the model shown here. Now the NPV of gross income realized, as shown, becomes a bit more than \$13M.

The gross income of \$13M has to pay for all other operational costs over those 9 years, including marketing and administration. Distribution costs, high for tangible goods, should be minimal for a web-service business. Only what is left over is the actual profit of the service business. During that period, the company is providing jobs and societal benefits.

6.2 Use of the Model

Having a model permits validation of the business model. Values can be balanced and ratios can be derived and compared with software business expectations. For Instance we find that over the 9-year period the cost to maintain the software approximates the allocated software share. The actual cost of maintenance over the 9 years is about \$4.5M, making the total cost for software about \$5.5M. These numbers indicate corresponds to a 82% maintenance to total software costs, a typical ratio for software in a volatile setting. By considering maintenance equivalent to COGS, it is paid out of ongoing income. During years 3 to 5 maintenance costs will be less than the expected 25% of gross income, later they greatly exceed that fraction. The actual total costs are approximately equal.

All of the assumptions made here can be challenged and improved in any specific situation. If results turn out poorly, prices may have to be raised. The effect of higher prices on the volume of services sold is an issue not covered here. Higher prices mean that some customers will be loath to use the services, the service may have to be fancier, salesmen will have to work harder, and also that income may be delayed. Business-level decision-making is required now. Those prognoses can be easily plugged into this software model.

If the service model is not based on charges to a service customer, but rather on related sales or advertising, hits then the pricing model changes drastically. The majority of customer services will not generate sales, so that software has to be configured that service costs for inquiries are very low. If there is a single or a few major customers, say a reservation service for an airline, prices and service levels will require negotiation, and having a model that allows assessing costs versus income is essential.

For all business models it is wise to assess alternate assumptions and arrive at ranges and explore what-if questions. All the values used in the assumptions have pragmatic ranges, and their leverage is clear.

6.3 Services versus other software businesses

While all substantial software requires maintenance efforts, in a service business such maintenance only contributes to attractiveness an ongoing use. When enterprise software

is sold, the customer also obtains a maintenance contract, typically for about 15% of the purchase price. That contract generates valuable income over the years at a low cost, the income becomes equal to the purchase price in less than 7 years [Wiederhold:03].

There seems to be no direct open-source business model here, since there is no income to compensate for ongoing operational expenses. Only if indirect income from related sales or advertising hits is obtained, can free services be provided. That mode is prevalent today.

Vendors of shrink-wrapped packages are able to sell later versions of those packages to their earlier customers, when to customer find that their current versions lack adaptations and perfections they need.

7. Conclusions

One conclusion is that profiting from software is not easy. Making assumptions is inherent in dealing with the future, and the alternative, remaining ignorant, provides even less guidance.

We cannot yet assess how the web-service business will develop. However, the software engineering experience that underlies the model presented here changes only slowly. In this model we ignore detail and focus on aggregate projections. Interaction with the overall business model is essential, but we often seen that the technical issues presented here are ignored. Maintaining software, although costly, is known to be essential. Maintenance provides continuing refreshment of the inherent IP and competitive stance of the business. Only with high quality maintenance a company can maintain an effective service model. The fact that accountants classify all software work as research and development gives the impression that maintenance efforts are optional, although they are as essential as engine power is to an airplane, even when it has reached the desired altitude.

Service models are very attractive to mature companies, that otherwise face the difficulties of always having to come up with new products to keep sales volume high. But operating in a service model means that management must appreciate the process and software staff must be motivated to sustain the value of the product by performing excellent maintenance, covering all of the three types presented in Section 3.

Technology and education favors novelty over maintenance. A typical and popular software engineering textbook devotes 3 out of 850 pages to maintainability and maintenance, although the same book states that the effort devoted to maintenance is greater than 60% [Pressman:01]. Many 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's of course an illusion that cheap labor reduces the cost of maintenance, it mainly reduces the benefits of maintenance. Managers often bemoan the high cost of maintenance in their organizations. If the company's management understood software economics they would let interns and recent graduates do innovative work, and reward the experienced staff who sustain the value of their existing products and services highly, but not by moving them to other tasks [Pfleeger:01]. This process has been referred to as a "gentrification" of maintenance [Foote:98].

A similar analysis is likely to be appropriate for services that provide content from databases, which have become a recent focus for protection. Those databases are also subject to continuing maintenance and improvements [GardnerR:98].

Even if the predictions made here depend on assumptions that cannot be verified in advance, having the assumptions stated explicitly allows discussions of alternatives. The spreadsheet used is available on my web pages [Wiederhold:95S]. Opinions and disagreements are brought down to a lower level of detail. Bringing the valuation methods into open discussion is an initial step.

Acknowledgements.

Discussions with a number of IRS and consulting economists helped in establishing the principles shown here. A course directed by Jeremy Dent helped in understanding valuation issues in general. Laurence Melloul provided references for information regarding Function Points. I received constructive feedback from John Wiederhold, Bhavani Thuraisingham, and Dan Grosshans. Any errors in this paper are of course, my responsibility, but I cannot be responsible for results obtained by using the presented model.

References cited.

- [Aron:83] J.D. Aron: *The System Development Process, The Programming Team (Part II)*; Addison-Wesley, 1983.
- [Barra.:98] BARRA: *United States Equity Version 3 (E3)*; BARRA, Berkeley, CA, 1998.
- [Basili:90] Victor Basili: "Viewing Maintenance as Reuse-Oriented Software Development"; *IEEE Software*, Vol.7 No.1, Jan. 1990, pp.19-25.
- [Becker:02] Brian C. Becker: Cost-Sharing Buy-Ins; Chapter A in Feinschreiber: *Transfer Pricing Handbook*, 3rd Edition, Wiley, 2002 supplement.
- [BeladyL:72] Laszlo Belady and M.M. Lehman: "An Introduction to Growth Dynamics"; in W. Freiberger (ed.), *Statistical Computer Performance Evaluation*, Academic Press, 1972.
- [Bernstein:03] Philip Bernstein: Remark at NLM/NSF planning meeting, Bethesda, MD, 3 Feb 2003.
- [Boehm:81] Barry Boehm: *Software Engineering Economics*; Prentice-Hall, 1981.
- [Boehm:00] Barry Boehm et al.: *Software Cost Estimation with Cocomo II*; Prentice-Hall, 2000.
- [Bonasia:02] J. Bonasia: Firms Wringing Value from IT Units; *Investors Business Daily*, 21 Aug. 2002.
- [BondS:01] Andy Bond and Jagan Sud: "Service Composition for Enterprise Programming"; *Proc. Working Conference on Complex and Dynamic Systems Architecture*, University of Queensland, Dec. 2001, Brisbane, Australia.
- [Brooks:95] Frederick Brooks: *The Mythical Man-Month, Essays in Software Engineering*; Addison-Wesley, 1975, reprinted 1995.

- [ChatfieldG:73] C. Chatfield and G.J. Goodhardt: A Consumer Purchasing Model with Erlang Interpurchase times; Journal of the American Statistical Association, Dec 1973, Vol. 68, pages 828-835.
- [Damodaran:2002] Aswath Damodaran: The Dark Side of Valuation: Valuing Old Tech, New Tech, and New Economy Companies; Prentice-Hall, 2002.
- [Foote:98] Brian Foote: Escape from the Spaghetti Jungle; Sprint Object-Oriented Users' Group, February 1998, <http://www.laputan.org/>.
- [GardnerR:98] William Gardner and Joseph Rosenbaum: Intellectual Property: Database Protection and Access to Information, *Science*, Vol. 281, Issue 5378, pages 786-787, 7 August 1998.
- [Gordon:82] Myron J. Gordon: *The Investment, Financing and Valuation of the Corporation*; Greenwood Press, 1982.
- [HennessyP:90] John Hennessy and David Patterson: *Computer Architecture*; Morgan Kaufman, 1990 (3rd Edition 2002).
- [IEEE:98] IEEE: *Standard for Software Maintenance*; IEEE Standards Library, Document 1219-1998, 1998.
- [Jones:98] T. Capers Jones: *Estimating Software Costs*; McGraw-Hill, 1998.
- [LientzS:80] B.P. Lientz and E.B. Swanson: *Software Maintenance Management*; Addison-Wesley, 1980.
- [Lim:98] Wayne C. Lim: *The Economics of Software Reuse*; Prentice-Hall, 1998.
- [Lipschutz:04] Robert P. Lipschutz: *A Better Blueprint for Business*; PC Magazine, September 7, 2004.
- [MahajanMW:00] Vijay Mahajan, Eitan Muller, and Yoram Wind (editors): *New-Product Diffusion Models*; International Series in Quantitative Marketing, Kluwer, 2000.
- [Marciniak:94] John J. Marcianak *Encyclopedia of Software Engineering*, Wiley, 1994.
- [Moore:95] [Geoffrey A. Moore](#): *Inside the Tornado: Marketing Strategies from Silicon Valley's Cutting Edge*; Harper Business, 1995.
- [Pfleeger:01] Shari Lawrence Pfleeger: *Software Engineering, Theory and Practice*, 2nd ed; Prentice-Hall, 2001.
- [Pigoski:97] Thomas M. Pigoski: *Practical Software Maintenance - Best Practices for Managing Your Software Investment*; IEEE Computer Society Press, 1997.
- [Pressman:01] Roger Pressman: *Software Engineering, A Practioner's Approach*; 5th edition; Mc GrawHill, 2001.
- [Rechtman:01] Ygal Rechtman: Accounting Treatment of Intangible assets; draft, <http://www.rechtman.com/acc692.htm>, July 2001.
- [Roux:97] David Roux: Statement about software growth; 1997.
- [SmithP:00] Gordon Smith and Russell Parr: *Valuation of Intellectual Property and Intangible Assets*, 3rd edition; Wiley 2000.
- [Stobbs:00] Gregory Stobbs: *Software Patents*; Aspen Law and Business publishers, 2000.

- [Wiederhold:83] Gio Wiederhold: *Database Design*; McGraw-Hill, 1983, also available in the ACM anthology, 2003.
- [Wiederhold:95] Gio Wiederhold: "Modeling and Software Maintenance"; in Michael P. Papazoglou (ed.): *OOER'95: Object-Oriented and Entity Relationship Modelling*; LNCS 1021, Springer Verlag, pages 1-20, Dec.1995.
- [WiederholdC:98] Gio Wiederhold and CS99I class: Web growth (updated); 1998/2000, <<http://www-db.stanford.edu/pub/gio/CS99I/2000/webgrowthslide.ppt>>.
- [Wiederhold:03] Gio Wiederhold: "The Product Flow Model"; Proc. 15th Conf. on Software Engineering and Knowledge Engineering (SEKE), Keynote 2, July 2003, Knowledge Systems Institute, Skokie, IL., pp. 183-186.
- [Wiederhold:05] Gio Wiederhold: What is Software Worth; manuscript for publication, available at <http://www-db.stanford.edu/pub/gio/inprogress.html#worth>, 2005.
- [Wiederhold:05S] Gio Wiederhold: Simple spreadsheets for Software Worth assessment; links at <http://www-db.stanford.edu/pub/gio/inprogress.html#worth>, 2005.

----- o ----- o -----