What is Your Software Worth?

© Gio Wiederhold
Stanford University

February 2007, last updated April 2008
A shorter version of this paper was published in the
Comm. of the ACM, Vol. 49 No.9, Sep. 2006 [Wiederhold:06].

Abstract

This article presents a method for valuing software, based on the income that use of that software is expected to generate in the future. It applies well known principles of intellectual property (IP) valuation, sales expectations, software maintenance, product growth, discounting to present value, and the like, always focusing on the specific issues that arise when the benefits of software are to be analyzed. An issue, not dealt with in the literature of valuing intangibles, is that software is continually upgraded. Applying depreciation schedules is the simple solution, but depreciation is taken by purchasers, and does not represent the actual diminution of the inherent IP of software at the supplier. A novel approach, which considers ongoing maintenance and its effects, is presented here. 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 alternatives. As an example we evaluate a service business model alternative. 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:81, 00]. Even software engineering management approaches termed `Earned Value Management' only deal with expenses within a development schedule [Abba:97]. While we, as software creators, believe that what we produce is valuable, we are rarely called upon to quantify its benefits [GarmusH:01]. One reason may be that much investment in software engineering has been motivated by military and governmental applications, where benefits are hard to quantify. A notable exception is a recent paper on measuring the value of IT assets [TillquistR:05]. 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 [Stobbs:00] [SmithP:00] [Lipschutz:04] [Bonasia:02]. The results are often inconsistent [Lev:01].

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 market for houses they design, a potter will know the price for the dishes to be sold, as will a builder of bicycles. These producers deal with tangible instances. But since software, once written, is easy to replicate at a negligible cost, each subsequent instance will be sold for much more than its incremental cost.

Potential sales volume is a key factor required to set a price which will provide an adequate future income. The initial value of the software to the creator depends on that income potential. Predicting the quantity of sales is hard, and expert help is often required. An author of a book can obtain guidance from a publisher. Once the book is on the market, the interested author is able to track what the total income is from the publisher's annual statements.

The value to the purchaser of a book or software 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 reads have little value. If creators are aware of the potential value of the product they will be better prepared to make decisions on the product design and the effort to be spent. 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 how software engineering is practiced.

1.2 Protection

There is substantial literature on the protection of the intellectual property value inherent in software. A dozen years ago an NRC study focused on copyright- and patent-based protection [BranscombEa:91]. Frank Ingari of Lotus complained in that report, that without understanding what is to be protected the focus on methods for protecting IP addressed a second-order question. Yet, copyright protection issues are still discussed widely today. In practice, copyright and patents afford only limited protection, since the methods to protect software are threatened in modern systems with rapid and anonymous communication. But the value lost by copying remains hard to ascertain, making it hard to balance protection versus openness [Gates:04]. Having a good product easily and early available for a reasonable price is the best strategy.

1.3 Who needs software IP valuations

It is certainly useful for software engineers to understand some of the economics of software use, rather than just being concerned about the cost of writing software. But being able to assign value to intangible property is assuming great importance in many arenas, 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. In 2003 the investment by U.S. enterprises in intangibles amounted to about 9% of GNP, versus an investment in tangibles slightly over 8% [Hulten:06]. Only a modest fraction of that a $1,000 Billion investment goes into software, since there are also investments made in other intangibles as marketing, business process improvements, job-related training, and other research, but software comprises still a substantial amount [Economist:06]. But software is so squishy, that traditional valuations methods, even those applied to other intellectual property as music, patents, and copyrights will give misleading results.

Important software arenas include:

1. Making business decisions about investing in software. Investment always has risks. For many industries 10% of their investments go into software. The fraction is higher in businesses as banking, less in healthcare and education.

2. Making decisions about acquiring software. Choices include purchasing software packages, leasing the use of software, obtaining maintenance for such software, or just obtaining software services from a web service company. While such decisions are mainly based on cost, understanding what one has obtained, and how long it will be of value, is crucial.
3. Making decisions about acquiring a software company. Part of performing so-called "due diligence" is assessing all the property being acquired, and valuing the software is a major part of that effort. During the dot.com bubble such properties were generally overvalued. Subsequently there was a reaction in the opposite direction, leading to massive writeoffs for acquired properties, and loss of confidence by shareholders.
4. Companies that export software to their foreign subsidiaries or contractors, perhaps as part of outsourcing should know what the value of their exports are, so they can put the right numbers into their books [Rottman:06]. The general problem is referred to as transfer pricing, but currently not handled consistently for software and related IP [WiederholdGM:06]. The rapid growth of globalization is bringing this issue to the forefront [Wiederhold:04].

While any method that requires predictions into the future will never be precise, having a consistent approach for these valuations will engender consistency. With consistency will come comparability, and a capability to adjust the parameters and the model so that precision can improve as experience is gained.

1.4 Outline

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 cost 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 overall project is profitable, or, if it seems not to be, 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 significant benefits to the supplier [Cusumano:04]. Over time maintenance costs typically exceed the original software development cost by factors of 2 to 10. Maintenance causes growth of software, and Section 3 will show how three types of maintenance efforts together affect software growth. 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 also some metrics. All of software engineering suffers from inadequate metrics, and valuation is no exception.

Section 5 contains the principal novel contribution for the IP analysis. It shows how growth and maintenance leads to a diminution of the value of the software over time, replacing alternate approaches based on research-and-development costs or on depreciation. In Section 6 we predict sales volumes for sold software. Fairly standard business methods are used, selected for their suitability for analyzing software sales. Finally, in Section 7 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 in Section 8 a service-oriented business model where maintenance income is included. The conclusion provides some advice, for individuals, managers, and educators.

This article brings together information from domains that rarely interact directly: software engineering, economics, business practice, and legal sources. I quote a number of rules of thumb. The references include some citations for 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 just some citations of recent topical articles, although many articles have helped to provide the insights presented here.

2. 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 [Damodaran:2002]. Tangible 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 such a company. Even today, the bookvalue of a company shown in its annual report is merely the sum of the value of its facilities, inventory, equipment, and finances. That bookvalue has little to do with how investors value companies in the software domain, where IP is dominant [Rechtman:01]. For example, we can look at SAP's Annual Report for 2003 and learn that its book value (assets - money owed) was about €6.3B. But using the market price of the shares (about €100) and the number of shares in circulation, about 315M, we find that SAPs shareholders valued the company at €31.5B. The investors in SAP base the value on the income they expect to obtain over time from their shares, and not on its tangible assets [Becker:02]. The difference, €25.2B, is due to the intangible property owned by SAP and its shareholders [Hall:99]. The value of the SAP brand in 2004 is estimated at €6.8B [BW:05]. Software, and the knowledge how to produce and sell it, comprise the rest, amounting to €18.4B. A dozen examples of U.S. high tech companies show similar ratios, but no routine annual report breaks down a corporate intangible value to the same detail that the much smaller bookvalue is documented [SmithP:00].

The intangible property owned by SAP, or any 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. In companies that use expert systems business knowledge can be directly incorporated into the software [FeigenbaumMNP:88].

The reputation of a software company can be increased by advertising and the value of its software can grow by spending on research and development. These two components are the considered intellectual property (IP) of a business, since the workforce of a company cannot be considered to be a property. Open-source software is also excluded. Valuation of IP is required when software product lines are purchased or transferred. When entire companies change hands, then the workforce can be assigned a value as well. Sometimes all intangible assets are thrown into a basket named `goodwill', but such a leaky container ignores the variety and importance of the intangible components. Although software is the most tangible of the intangibles owned by businesses, its valuation is still poorly understood, leading to a gamut of problems [Lev:01].

2.1 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 [SmithP:00].
This article focuses only on software, likely the largest component of IP owned by companies in the field of computing. Ownership of software is not limited to companies that produce software as a product. The majority of modern businesses create, purchase, maintain, and benefit from software. Banks could not exist without software; there is no other way to determine what is due to a customer or what the customer owes. In those settings software intangibles are valued by the sales of tangible goods they motivate or enable. 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. I am certain you can create a similar scenario for your own industry.

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 distinguish now software producers and software users. In the US, software sold for to users amounted to about $120B in the year 2000, about half for prepackaged software and the other half for custom software. Another $128B was invested by companies for internal software development, both for new software and maintenance [HandL:03]. What is income for one segment of industry is an investment for another segment, but in the end there should be an economic benefit gained from the use of all that software.

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 is different even 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 of software to keep some existing customer creates a net loss for the seller, while being out-of-sync creates inefficiencies for the customer. Furthermore, the new version, if adequately debugged, should be substantially better than the prior version. For the purchaser a deal on obsolete software is likely to create expenses in adaptation and integration. Sometimes an old version of software has to be supported to keep some existing customer who cannot or will not update, but that support creates a net loss for the seller and being a year or so out-of-sync with ones peers creates inefficiencies for the customer. Since a new version of the software product includes much of the code and all of the functionality of the prior version, IP from the prior version continues to contribute to that new version. Disasters have occurred when new versions did not replicate all prior functionalities [Splosky:04]. Fundamental parts of software can easily live 10-15 years and hence continue to contribute to the generation of revenue. We will deal with that aspect in Section 5.

In companies that use software, the valuation must be based in its contribution to the income of the company. In early days, one could compare a company's operation prior to using software and subsequent to installation, and assess the benefits of software based on the difference [Batelle:73]. The value would be based on an increase of productivity, that is how many more goods were produced and how much production costs of the goods sold (CoGS) were reduced. For consulting companies productivity is in terms of services and the capability and cost of the workforce has an equivalent role. The improvements brought about by automation, after considering its cost, provide a measure of income attributable to software for that year. Future years would bring in similar or higher benefits, as the organizations adjusted to new tasks and workflow. The life and the ongoing cost of maintaining the software still has to be estimated, and we'll do that in Section 5 as well.

Today it is rare that a broad set of new software applications will be installed within an ongoing company. More commonly, upgraded versions of existing software will be obtained, or some poorly served aspect of the company will be automated. At times a major subsystem will be substituted [TamaiT:927]. Major substitutions will be rare after the Y2K bulge, when fear of serious operational problems motivated much scrapping of obsolete software. Comparing a business that has adopted a significant improvement, say on-line ordering, to a similar business that has not yet converted can provide input to assess the benefits that are due to the additional software. However finding comparables is hard, and invariably adjustments will be needed before incomes can be compared.

In large integrated companies it becomes impossible to relate income directly to software applications. Many resources are employed to generate income. There are routine operational costs as well as `Intellectual property Generating Expenses' (IGEs), such as software development and maintenance, advertising for new products and corporate recognition, investments in quality control, and the like. An approach that remains is based on a belief that the management of a company is efficient in the allocation of its resources [Samuelson:83].

Pareto Rule: ***At the optimum each investment dollar spent generates the same benefit***

This rule is based on an assumption of achieving optimal spending and convex benefit to cost curves. Having a convex relationship, often expressed as a parabola, guarantees that there will be single optimum. Deviating from that optimum may increase income or reduce costs, but the total effect will be negative. Now all the income created by all expenses, IGE and personnel directly involved in income generation, can be allocated according to the proportion of costs incurred. (Note: this is not the other Pareto Rule on 80/20 instance/effort allocation).

Under Pareto optimal conditions spending $100 000 on personnel should deliver the same growth in net income (sales revenue - cost of sales) as spending $100 000 on software. We can then allocate a fraction of the company's net income to software that is proportionate to its fraction within all the expenses that generate income. Overhead will be excluded. There will be variation in that fraction from year to year, but over the life of long-lived software such variations even out. If a company behaves very irrationally in its spending on IGE and personnel, more so than its peers, it is bound to have lower net profits, and its IP and its stockholders will suffer as a result.

We noted earlier that income-based measures don't work in governmental and military settings. In those organizations measures of productivity and the extent of cost-avoidance have to be combined to produce a surrogate for income. Comparing the size of the workforce employed without and with software for affected tasks provides the most valid alternative. In the military however, there is also much high-cost equipment, which, in turn depends on software. In those settings, and in other non-profit institutions, as academia, using an assumption of rational behavior for relative allocation is even more questionable. Valuations of the effect of software will hence be quite inexact, and mainly of comparative use.

2.3 Revenue and gross profit

In business financial reporting the revenue realized is immediately reduced by the cost of the goods sold. Here is where software and much other intellectual property differ from tangible products. The effort to make the first unit of a product is a major cost in both cases, but tangible goods incur a manufacturing cost for each successive unit, while for software and many other intangibles the manufacturing cost is negligible. If software distribution is over the Internet, there are no direct 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 (gross profit/revenue), are close to one and essentially meaningless. One cannot base investment decision on income margins that are, say, 98.5% versus 97.2% of sales. These ratios obviously do not represent reality over any period of time. We make a suggestion about this problem in the conclusion.

If we were to amortize initial research and development costs, as well as ongoing maintenance costs, over each unit of product and include them in the cost-of-goods sold, then those margins could become more meaningful. However, without predicting future sales, the initial costs cannot be allocated to units of products. In IP assessments those prior costs are ignored. Once the value of the software beyond today has been determined, then one can decide if those earlier investments were worthwhile.

Since we only assess here the value of existing software, we ignore its initial research and development cost. We also ignore its negligible manufacturing cost. Now the income per unit is equal to the revenue, i.e., the price it fetches and the sales volume. However, there will be ongoing costs to keep the software viable for sales. That distinction is recognized by accounting rules: costs prior to showing that the software is feasible, are to be expensed, development costs beyond that point are to be capitalized [SmithP:00]. No mention is made of maintenance costs, which would be more properly accounted as cost-of-goods-sold, since they sustain the ongoing viability of the product.

The next section addresses the issues that occur because maintained software is so slithery. Software keeps changing while one tries to understand and measure it. If software were stable, it would act like a tangible product with low manufacturing cost: "Hardware is petrified software" [Panetta:89].

3. Sustaining Software

Before we can proceed moving from the income generated by software to the 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 little software beyond basic mathematical libraries is stable [ReillyS:98].

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 [Spolsky:04].

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 its contribution will have to be deducted in the valuation. In order to be able to quantify that deduction we summarize a prior business analysis [Wiederhold:03]. In Section 8 we consider an alternative which includes the benefits of maintenance, which then must also consider the cost of such maintenance.

Successful software products have many versions, long lifetimes, and corresponding high maintenance cost ratios over their lifetime. Software lifetimes before complete product (not version) substitution 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 when software is new versions may be issued more frequently, while the rate reduces later in its life.

Maintenance costs of such enterprise software amount to 60% to 90% of total costs [Pigoski:97]. Military software is at the low end of the maintenance cost range, but that seems to be because users of military software users can't complain much, and most of the feedback they generate is ignored. The effect is that military software is poorly maintained, and requires periodic wholesale substitution [Hendler:02].

3.1 Continuing improvement

We use well-established definitions for the three classes of long-term maintenance which improve the product [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]. Collecting and responding to feedback, crucial to IP generation, is detailed in Section 3.2

1. **Bug fixing** or **corrective maintenance** is essential to keep customers. In practice, most required 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 will 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. Adaptations allow the software to deal with new hardware, operating systems, network, browser updates, as well as other software 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 improvements as 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, 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]. Military software, rarely perfected, wastes much user effort, and hence rates low on a benefit/cost scale [Hendler:02].

Bug fixing, for software that is accepted in the market, eventually reduces to less than 10% of the maintenance effort [LientzS:80]. Adaptation consumes 15% to 60% of the maintenance costs [Glass:03]. The effort needed varies with the number of interfaces that have to be maintained. Ongoing perfection consumes about half of maintenance costs [Pfleeger:01]. Marketing staff often touts the results of perfective maintenance as being novel and innovative, even when base functionality does not change.

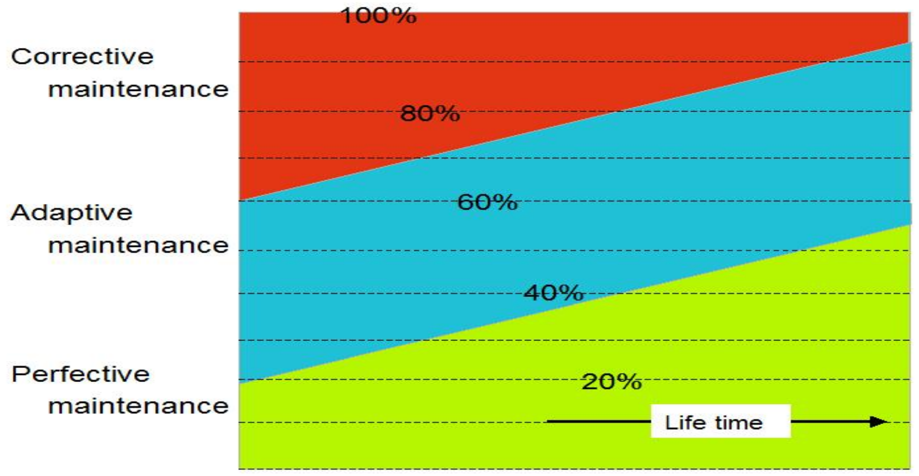

Figure 1: Maintenance Effort over the Lifetime of software

The effectiveness of maintenance is greatly hindered by poor design and lack of adequate documentation [BeladyL:72]. A simple mathematical model promotes use of a common architecture for interacting applications to lower maintenance costs [Mookerjee:05]. Well-designed systems will be improved more rapidly, and hence, paradoxically, consume more maintenance [Glass:03]. But those systems will also have a longer life, while unmaintained systems will die [Parnas:94]. Figure 1 depicts the relative effort distribution over time; in practice the ratios will differ depending on the setting and on the impact and timing of external events.

There are several related aspects that motivate a supplier to maintain a product at a high level. With every version the product will be better, and be easier to sell. If a maintenance fee is charged, ongoing maintenance is expected and essential to keeping costumers. Then those fees contribute a stable income. It is easier to keep an old customer than to gain a new one. Good maintenance has a high value to the customer and can generate substantial revenue [Wiederhold:03]. Maintenance fees contribute in time as much revenue as new sale licenses.

3.2 Sources of IP in the Maintenance Phase

Maintenance is too often viewed as work requiring little expertise. That view is misleading. While the coding may not be that innovative, responding effectively to the diverse demands for maintenance is a real challenge [Parnas:94]. The diverse demands manifest themselves in the diversity of information sources. Those sources contribute to the incremental IP as much as the subsequent coding.

Corrective maintenance is based on feedback from the users and customers. A specialized team at the supplier 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. 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 required to sustain supplied software so it keeps up with external changes of its settings. Input for required adaptations comes from standards bodies, from hardware vendors, from software vendors who make products that interface with the software, and from government sources. Major software suppliers have representatives at many external organizations and vendors, others will follow their lead. Filtering and scheduling adaptations to keep up with customers' needs 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 throughout the entire organization 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.

There will always be new uses for software that, with some changes existing software, can cater to. Information from responsive sales staff is the best input. Review of feedback obtained after delivery is important, although frequently ignored [Glass:03]. Perfecting updates should not disturb current users, a constraint that becomes more difficult to satisfy as the software is used more broadly [Aron:83]. During perfective maintenance user inputs dominate technical concerns [NodderWG:99]. Code that considers the many 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 contribution to growth.

The inputs that lead to maintenance actions represent intellectual input not present in the original product. They hence should not be valued as part of the original IP.

3.3 Alternatives for Assessing Dynamic Software IP.

There are two fundamental alternatives of assessing the contribution of the research, development, and maintenance efforts over time to the value of the software.

1. Accumulating all the inputs and aggregating their value over time
2. Measuring the output, namely the software itself

For the first alternative one must make assumptions about the value of research and development input over its life. For patented inventions the value would have the lifetime of the patent, for other contributions the life would depend on the degree of protection through trade secrets and copyrights that are available. If some software was acquired its cost is added as well. All inputs prior to the point that the software becomes available can be aggregated over the remaining life of the research contribution are aggregated over the future years. Earlier research contributions are termed spillover, and the delays, or lags before inputs can generate income have to be considered [LeonardS:05]. Use of this method requires careful accounting of all prior inputs, assumptions about the life of the contributions, and the use of factors that relate the expected leverage of various types of research, development, and maintenance to income.
This method requires much data, data that is rarely gathered while software is being developed.

The second method measures the output, namely the software that is created by the efforts. That software can be measured in a variety of ways, but assumptions will be needed to relate the product to its value. The remainder of this exposition uses the output approach, focusing on the software that is generated. However, this method only works for existing software, and cannot be applied to value early-stage concepts or prototypes.

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 measure what happens to the code and its functionality. Most software metrics have been developed to deal with development cost, productivity, and quality, and must be re-evaluated to deal with valuation [Gilb:05].

4.1 Metrics

Two primary metrics are in use for estimating code size and effort: Lines-of-code (LoC) and Function Points. Both have problems, but they also complement each other.

The size of programs tends to be easy to measure once the software is completed. One can look at lines-of-source-code or the volume of binary code. Some adjustments are needed in either case. When counting lines-of-source-code (SLoC) comment lines should be ignored, although they have value for future maintainers. If multiple statements appear on a line they should be counted distinctly [Park:92]. Any software modules that appear multiple times are counted only once, since that is an appropriate measure for the IP. If modules have been written in multiple languages, then the lines cannot be directly aggregated [Kemerer:93]. Tools that can process a variety of languages are publicly available [Wheeler:05].

Function points (FP) provide a useful measure of potential code complexity before programming starts. FP counts are based on the numbers and sizes of inputs and outputs being specified. The FP literature has provided tables that relate FPs to code for many languages [Jones:96]. Common procedural languages have ratios of about 100 SLoC per FP, while Visual Basic is rated at 35 SLoC/FP, allowing a normalization.

Formulas used for FP-based effort estimation recognize the non-linearity of efforts that makes large programs so much more costly to manage, write, and test. It provides adjustments factors that can account for unusual algorithmic complexity, for programming languages, and for staff competence. FP-based estimation is supported by empirical formulas that will predict programming effort and code sizes. However, the FP metric is insensitive to the incremental, but valuable improvements made during maintenance. Also, since FP metrics are intended to estimate initial development it is rare that updated FP-counts are maintained as new versions are created. To overcome that lack, a technique termed backfiring attempts to compute FPs from SLoC data [Jones:95].

Binary code sizes are harder to normalize for comparison. Again, modules should be counted only once, although they may occur repeatedly in distinct load configurations. Compilers for some languages create quite dense codes. Other high-level languages rely greatly on libraries and bring in large bodies of code that may not be relevant and do not represent IP. Normalizations may be needed for binary codes as well, but there is not the same body of experience that exists for source code. Conversions to source lines has also been attempted for the C language [Hatton:05]. Frequency of module use can be measured dynamically, but may not reflect at all the relative inherent IP value.

4.2 Relative Value of Old and New Code

We measure code sizes to allocate its relative contribution to IP. The 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. Even though this metric is indirect, it serves the purpose here, namely to have 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 is that the basic functionality 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, which are critical to the IP. The architectural component of software also represents valuable IP and changes little over the life of the software.

We find indeed that much 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 of new code. We feel comfortable using relative code size as surrogate to measure the IP in a piece of 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.

4.3 Growth of code

The maintenance activities that sustain software cause the software to grow in size, as presented in Section 3. Hennessy and Patterson 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 implies 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.***

Making many changes to produce a new version will allow too many interactions among them. A growth larger than 30% will make the a version of a product unreliable [Bernstein:03]. Cost estimation tables further support such barriers [Jones:98]. The existence of a limit due to complexity 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]. We were able to validate a rule suggested by David Roux which defined a more modest, rate of growth [Roux:97]:

DR rule: ***Software grows at each version equal to the size of the first working release.***

A linear behavior was observed for operating system modules, as well as its effect on maintenance costs and reliability [BeladyL:72], [Blum:03], [McGraw:03]. That behavior has also been independently validated [Tamai:92]. In embedded systems, for instance game software, where typically severe memory constraints remain, that rule cannot be applied.

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, then 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. It's a common aphorism that only the version 3 of a software product is truly useful, by then software growth obeys Bernstein's limit [NodderWG:99]. In the early phases of a product a higher rate of growth is certainly possible; most of the original developers will still be on board, there is good understanding of the code, feedback from the field can be rapidly accommodated, and early adopters will appreciate improvements, so exceeding the PB rule limit initially seems acceptable. If the valuation starts with a later version, i.e., more mature code, then the subsequent growth will not appear to be so rapid.

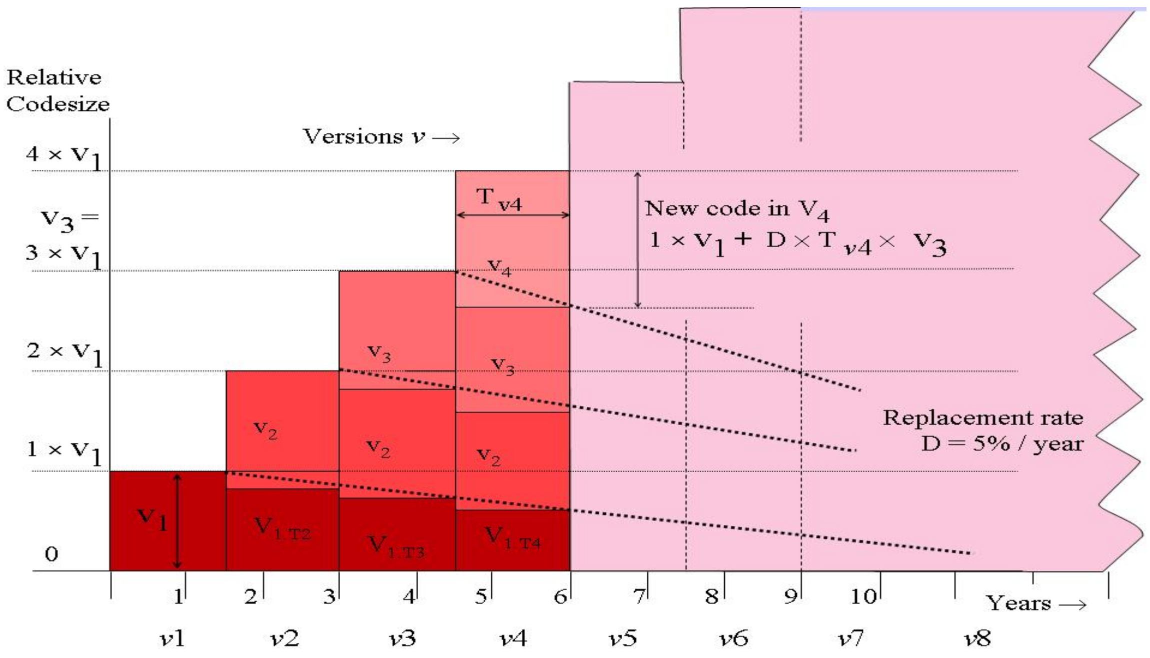

Figure 2. Code growth and reduction of earlier contributions over time.

For simplicity we consider the IP represented by remaining code from Version 1 through the years. 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 version release horizon. We adjust the results to the end of each year in Table 1, since income is counted at the end of each year.

4.4 Code Scavenging

During maintenance some code is 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. Outside of embedded systems 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 user's program may still depend on superseded or 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 8% for code that is aggressively maintained.

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

The limit of removal of code for a new version, even after several years, appears to be 20% to 25%. Beyond that level is better to rewrite the code [Glass:03]. However, rewriting the 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.

For simplicity 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. We consider all new IP to be represented by new code, and focus now only on IP retention represented by old code.

4.5 Cost of Maintenance

The cost of maintenance is typically greater than apparent from the growth of the code. While the code grows linearly by version, the personnel required to maintain the code will grow more rapidly. Brooks hypothesized a labor cost growth that was exponential with the size of the code, because of the rapid growth of code interactions [Brooks:95]. With a good software architecture, careful modularization, and principles as object-oriented design, the effort being expended may grow less dramatically, but is still likely to grow more than linear,

A factor that will have an even greater effect, but is nearly impossible to measure, is the management and the quality of personnel performing the maintenance [Maraia:05]. If the original authors of the code are gone, maybe because they have been promoted for doing such a good job, the novices that now must maintain the code will be less efficient and are also likely to introduce errors because of misunderstandings. The cost factor due to having inexperienced personnel performing maintenance will be very high.

The result is that personnel effort spent on maintenance is not a good surrogate for the incremental value of the work. Using the size of the code as a metric, as done in Section 4.2 is more reasonable, but will not adequately reflect the incremental costs incurred in practice.

5. Diminution of Software IP

To relate code sizes to IP value we observe that the price of the software remains constant, although the code representing that software grows steadily. We can then allocate that income over time.

5.1 Constant price assumption

Although the body of code grows and grows over time, the price of a unit of software sold tends to stay quite stable, typically increasing less than the rate of inflation. There are several reasons here. From the customer's point of view, a new version does not add functionality, it only provides the scope, reliability, and ease of use that should have been available in the first place. Income for software on long-term maintenance contracts does not benefit from price increases.

P: *The price of software for the same functionality is constant, even if it is improved.*

If actual prices are known to increase, corrections can be made. But now corrections for inflation have to be applied as well. Actual prices are also manipulated to gain market share, with the expectation that initial losses will be offset by grater gains in the future.

If a dominant position for a product can be achieved, then the vendor gains pricing power and the unit price can be raised to compensate for earlier losses. But even major vendors recognize their limits. A vendor who raises prices for software that is now in the public view and functionally well understood provides an incentive for imitators [Gates:98]. Keeping the price low for a comprehensive and reliable product discourages entry of competition.

Some products are sold via different channels, as retail and directly to Original Equipment Manufacturers (OEM), who bundle the product into their system offerings. If there are multiple channels the initial and future mix of sales channels has to be considered to obtain valid future prices.

Overall, keeping the model as simple as possible makes it more reliable, a principle known as Occam's razor [Heylighen:97]. Surveys of many software products show amazing price stability once inflation corrections are made, even while their capabilities and sizes increased manifold.

5.2 Income from a software product over time

We consider two classes of software below, enterprise software, such as databases and application tools built on top of them, and shrink-wrapped software. For software developed in-house it is harder to discern clear strategies.

Accepting a constant expected price v_1 allows us to rescale Figure 2. We see now how the rapid growth of software, especially initially, reduces the value of the initial IP contribution to the 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, but in Section 4.2 we determined that to realize income from that value much further work is required throughout the life of the product, generating the `New Code' shown in Figure 3. In Section 7 we present an alternative model, a model which considers the value of ongoing maintenance as well as the cost.

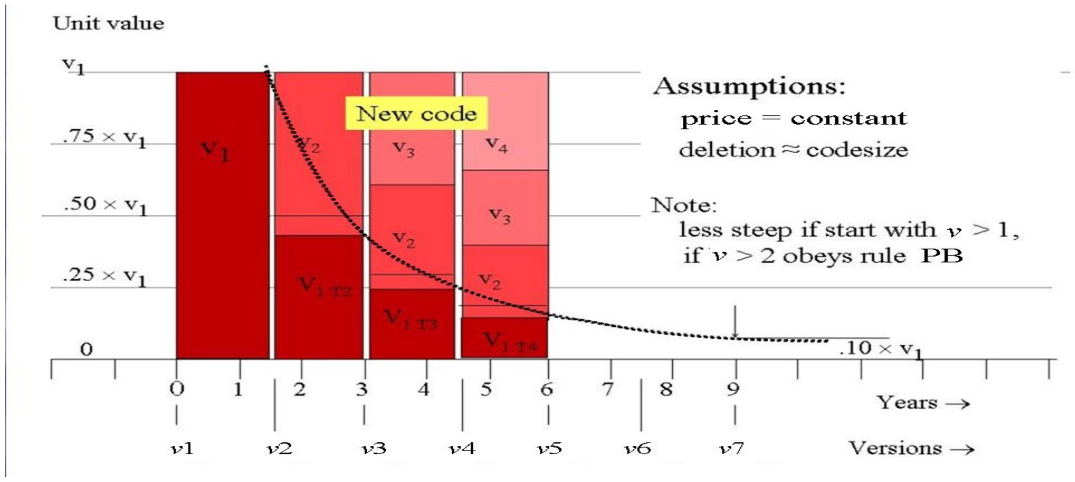

Figure 3. Diminution of the original IP due to maintenance..

Maintaining Enterprise Software. The common strategy for providers of enterprise software is to commit themselves to deliver any further versions of the software to existing customers, as long as the 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 and avoid the seller's obligation to fix problems in older versions. A larger fraction goes to routine call-center support. The remainder of the support fees, amounting to 6 to 10% of the purchase price in every subsequent year, is available to the engineering staff for the types of maintenance presented in Section 3. That income will also support most improvements needed to attract new customers to an existing product line.

Maintaining Shrink-wrapped Software. For shrink-wrapped software the marketing strategy differs. A customer is sold a version of a product, say for $500, and the customer is now motivated to keep it as long as it is adequate. When a new version is available the producer will make an effort to get the customer to substitute the upgraded version for the version in use.. The market literature will stress the innovation in the new version, but cannot afford to advertise it as new invention [Polk:06]. The new version is typically priced at the same level, unless very significant new functionality is provided. We ignore here the cost and benefits of truly new functionality, since our objective is to assess the IP inherent in the original product. For a similar product, with higher reliability, better adaptability and some perfection, the price tends not be greater than the rate of inflation. To motivate acceptance, existing customers can often obtain the upgrade at perhaps half of the new price. But it is also easy for the customer to skip some of the upgrades. When skipping versions, in time, incompatibilities due to missing adaptations become excessive, and a new version must be purchased. In this strategy of selling substitute versions the sales costs incurred by the vendor are high, so that net income is less. For upgrades at half price the net revenues may be 30% of the original purchase price every three years. Since customers without maintenance agreements have low expectations for call center support etc., those expenses tend to be low. In the end, for companies using a version substitution strategy the annual amount available to the engineering staff may again be about 6 to 10% of the original purchase price a year, although I have not been able to cross check such numbers.

In the end, the income and cost models are remarkably similar for enterprise and shrink-wrapped software. In-house software development will follow a different model, and Section 6, dealing with sales volumes, is not applicable. If developed software is only used in-house, then internal growth of usage and the growth of the total enterprise provides an equivalent measure.

6.0 Revenue from Software

If the income per unit of software sold is constant, then any income increases are due to growth of unit sales. For software being sold, sales can increase until its market is substantially saturated [Porter:98].

The size of the candidate market can be estimated in a variety of ways. Multiple approaches should be used to help reduce the uncertainty. Common ways of estimating future sales volumes include using the sales data about a predecessor product, the number of businesses that need the functionality of the new product, the number of customers that can afford the product, the number of a certain type of computer or operating system in use, and similar bounds. Deciding what the types of customers are not only helps in assessing their numbers, but also the types of features that a product requires. If the product can only attract technologists it will need more features than a product that is meant for a broader audience. A product for a larger, but more general audience should be easier to use, and should have fewer confusing features.

6.1 Penetration

A 50% market penetration 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. In practice there is always limit to growth. For any product being sold, a final limit to its sales growth in the future will be the growth of the entire world economy: *An unsustainable trend cannot be sustained* [Stein:74]. But much smaller constraints should be envisaged for nearly any business venture.

Via the Internet a truly useful and well marketed product can become rapidly known to the customer community. The ease of distribution of software means that a product can be rapidly delivered to all customers, so that substantial penetration is achievable in a short time. Further growth occurs when early adopters spread the news about the product's benefits [Moore:95]. If software installation requires a major effort or has perceived risks, the sales peak will be delayed. To sell yet more, successor products, incorporating true product innovation are needed, but that issue is not part of our discussion here.

Section 5.2 already accounted for the benefits of sales of substituting versions, so now the issue is new sales. There is some, but little definite literature on sales expectations [MahajanMW:00]. Some of that literature stresses the benefit of novelty, and if a piece of software derives its value from novelty, then use of a temporally limited model is appropriate [Moore:95]. Moore uses normal curves to describe the range of customers from early adopters to broad acceptance to laggards, as shown in Figure 4. That model is simple, requiring only a mean and a variance as input, but a problem with a normal distribution is that it begins far in the past, so that the actual point where sales and income commence is hard to determine.

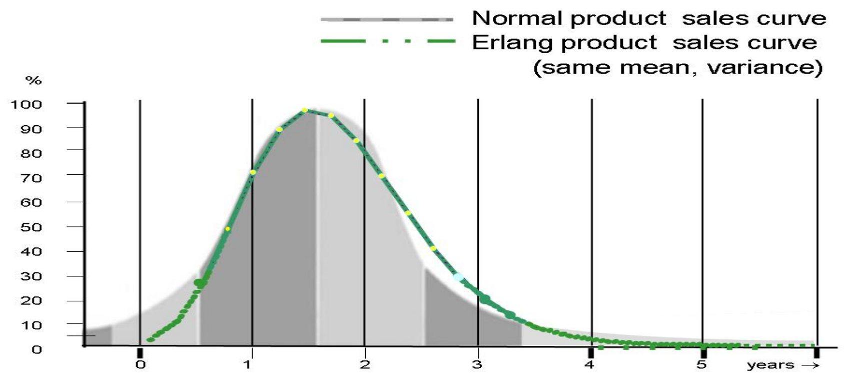

Figure 4. Normal and Erlang sales distributions.

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, but is foreshortened and has a maximum rate before its midpoint. We show in Figure 5 such curves for sales of about 50 000 units over a product's life, including distributions corresponding to small and large 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 and implicitly a high value of novelty; 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 means 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.

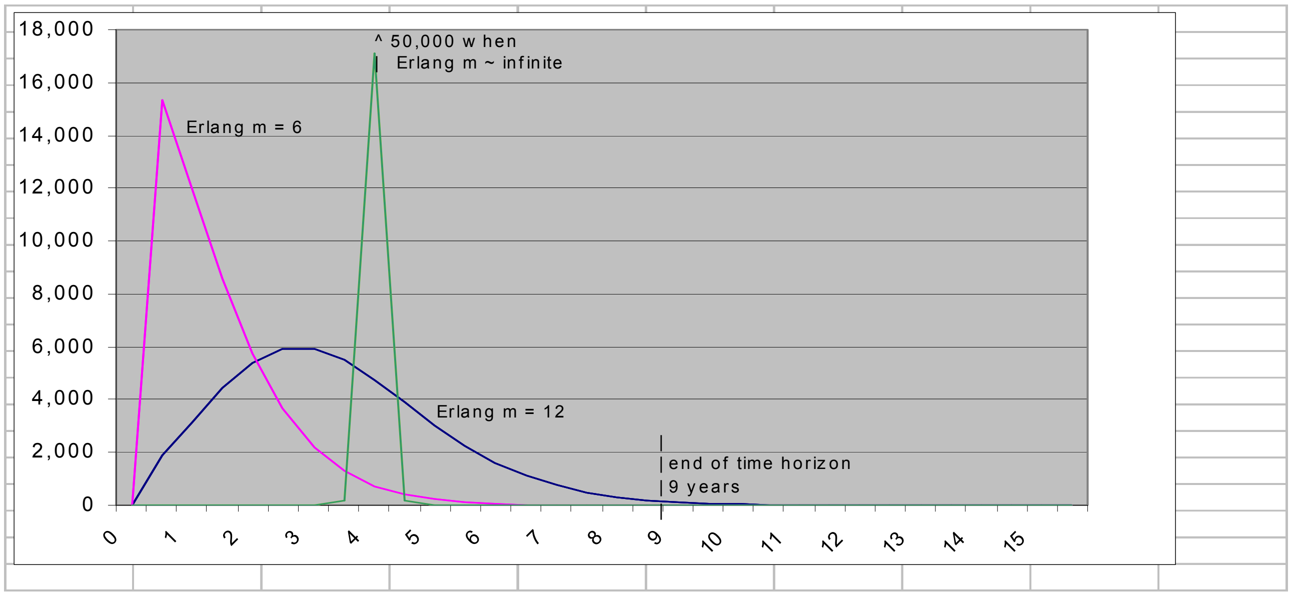

Figure 5. Semi-annual sales for an expected 50,000 units over 9 years, and continuing.

Sales based on some fixed IP can never continue forever. Eventually the penetration is as deep as it can go, and then only substitutions would be sold. It is essential to perform a reality check by taking the aggregate of expected new sales over time, and checking if the expectations are valid. Since actual sales and income from sales lag behind development expenditures, in our model of Table 1 we start counting income after one year. Lag is discussed in more detail in Section 8.3.

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. The rise, fall, and recovery of the dot.com business caused many projections to be faulty, assessing end-user markets at that time would have been useful. In a freshman class of 1998 we considered the expectations for on-line toy-sales and came to the conclusion that by the year 2003 more toys would be sold on the web than the total number of toys being sold, an unlikely situation [WiederholdC:98]. Such simple checking of limits and identifying inconsistencies can prevent many problems due to excessive expectations. For instance, infrastructure companies, as Cisco and other communication companies, that took in orders from enthusiastic starting dot.com operations could have realized that the total equipment being ordered was greater than the world could absorb.

We only consider annual sales. Our model does not account for seasonal variations. The effect of seasonal sales will differ by product type, being greatest for consumer end products and less for professional products. There is also an effect triggered by version announcements or rumors. Since installing software takes customer time and has some risks, when a new version is announced sales will diminish. When the new version appears, sales will increase again to compensate for the prior loss.

6.2 Expense attribution to software versus other business elements

Revenue from sales is first reduced by the cost of goods sold, a negligible fraction for pure software companies, so that gross profit is nearly the same as revenue from sales. Still, businesses do have operational costs that must be supported by the gross profit: software maintenance, marketing, advertising, sales staff, administration, interest for working capital, etc. Since without these costs no revenue will be generated, these costs reduce the benefits of the initial software.

Cost of Capital. The shareholders provided the initial capital. Once the company can sell its products those investors will also want their share of the pie, either as dividends or plowed back as reinvestments to grow the company. On any profit distribution, as dividends, income taxes will have to be paid. To avoid paying taxes, it is better to minimize profits and invest in company growth, as long as valuable improvements or innovations can be identified. Once a company is profitable, loans can be used to pay for growth, since interest on loan payments is tax-deductible.

Low capital requirements are typical for the software industry. Venture investors paid for the initial investment in people for planning, design, and development. They will profit when they sell their shares.

In general, a company has to decide to what extent to reinvest the profit, giving the stockholders increased share value, or pay dividends, giving the shareholders immediate income. In our model, which approximates many software companies, all available funds are used for further software development and advertising, and no taxable profit is generated.

Marketing Costs. When marketing costs have to analyzed, two types of expenditures have to be considered. There are sort range advertising costs directly associated with product sales. They will be intense early on, and then can often diminish, once the software is known. Major expenses need not extend beyond the initial, steep part of the Erlang sales curve shown in Figure 5. The other type of advertising focuses on enhancing the brandname of the entire company. Here the effect is intended to span products, so that those expenses should continue over the long range. Companies use very different balances of product and brand type advertising. Still, when advertising is stopped, its effect diminishes rapidly, and the reputation of the product will be the main factor for continuing sales. That product reputation is a result of effective technological investments, no amount of advertising can overcome a reputation of having poor quality in the software arena.

General and Administrative Costs. A substantial fraction of the expenses of doing business is hard to allocate. All those costs are dropped into that G&A category. We find substantial differences how these costs are handled in the financial statements of public companies, as annual reports or published 10-K statements. For instance, sometimes costs incurred in selling and distributing products are assigned to cost-of-goods sold, a good approach if a reliable relationship exists. But, as discussed in Section 3.2, sales personnel can contribute greatly to product improvement and marketing, especially if the service model, introduced in Section 8, is adopted.

Low distribution costs are also typical for software, especially if Internet capabilities can be exploited. To gain visibility however, substantial advertising may be needed, so that marketing costs replace distribution costs.

Costs to be attributed to pure management and administration are often small in startups. Those costs are only indirectly related to revenue. Allocating 20% of all costs provides a guideline.

Software R&D Costs. All software development is initially booked as ongoing expenses. We will see that substantial costs go into maintenance, The remainder of the software budget is used to develop new products, generating new intellectual property and hence new value to the stockholders, and is a surrogate for profit.

FASB rules state that software expenses after a product is brought to working state, say at a beta release, should be capitalized, and the resulting capital growth must then be depreciated, but that rule is often ignored [Lev:04]. Once a company achieves steady state, the combined effect of ongoing capitalization of software cost and its depreciation is minor.

Sources of Information. The information for our analyses is available in the financial statements of comparable public companies. Venture capitalists have insights into ratios appropriate for startups. Some summaries appear in books that give practical valuation and licensing guidance [Razgaitis:03].

For our example company we assign typical values for the software industry:

1. 5% to cost of capital - interest on loans, no dividends on stock
2. 45% to administrative costs, including sales and distribution

3. 25% to software development, shown as research and development
4. 25% to marketing and advertising.

In many businesses depreciation of tangibles and taxes greatly reduce net profit, but since there are few tangibles and taxes in our sample company, we ignore those items. Our sample company invests in future growth. The investors, being stockholders, expect to benefit in the long term from increased share prices. 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 Section 7.

There are metrics in common use to see rapidly if companies are healthy compared to their peers. Not all ratios make sense, as indicated earlier, in companies with few tangibles and a negligible cost-of -goods-sold.

6.3 Income Attributable to Software

Sales provide the revenue needed for software businesses. To see what fraction of the revenue contributes to IP one must separate out the R&D expenses, advertising costs, any growth of book value, and any dividends and bonuses paid out to stockholders. Taking all of these at equal value makes the assumption that management is smart, and will spend its money where it does the most good, the principle expressed in Section 2.2 as Pareto optimality. There can be much variance in these numbers, so to be credible data should be aggregated over several years and validated by comparison with similar businesses. Note that software expenses are included, they provide an investment in the future of the company, and will enhance its value.

We consider 4 types of companies, using income strategies described in Section 2.2.

1. Companies that develop and sell software
2. Companies that purchase and license software for internal use
3. Consulting companies that develop software internally for their own use.
4. Manufacturing companies that develop software internally for their own use.

We ignore the prior research and development (R&D) investment since the IP assessment focuses only on future income.

1. For a software developer we assume in our simple model that all available income is reinvested in the software business. No dividends are paid to stock holders and no significant capital expenses are incurred. The software investment includes here the ongoing research and development (R&D) expenses applied to fixing, adapting, and perfecting existing software. Such maintenance creates more code, as described in Section 4.4, and doesn't represent the original code we are valuing. It does represent a positive investment in the future.

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-month recommendations about quality software. After all expenses, a typical software developer can reinvest about 25% of total revenues into software research, development, and maintenance. That fraction is then the total contribution to IP.

2. Companies that purchase software licenses for all their needs have decided that their corporate value is not due to the uniqueness of their software -- they have essentially outsourced that aspect of their business. Software will contribute to their operation, but create only routine value. These companies are of great interest to the sellers of software, because effective use of purchased software leverages the IP inherent in the product. But the purchaser's IP is not due to the software, but due their organization and ability to use it. Costs for company-specific selection and integration of purchased software could be considered contributions for the allocation of net revenue.

3. Consulting companies do not generate tangible goods. Their revenue is generated primarily by their personnel and the tools used by their people. If they build all of their software in house then one can assign an equal benefit to software creating employees and other creative employees. Spitting the revenue according to that ratio provides the revenue to be allocated to software. If unique software is built to order by contractors its cost can added to the cost of the software staff. The future income for the consultancy must then be estimated based on past growth and comparable companies. The effective benefit of perpetual income can be estimated using the Gordon Growth Model [Gordon62]; a 15-year life gives roughly the same value at typical discount rates [BenningaS:97].
4. Manufacturers of tangible goods also rely greatly on software. But now there is a real cost-of-goods that has to be deducted first. For businesses that mainly produce tangibles the income that can be attributed to IP within the products tends to be lower, a ratio of 15% is typical in settings where there is competition and the software contributes to product differentiation. A smaller percentage of the staff than in the prior cases will be contributing to long-term IP, but even not all the IP is related to software, but also includes design and marketing. To allocate the software portion of the total IP one can again determine the ratio of software expenses to all business investments that are not directly related to manufacturing of products. If the software could alternatively be obtained from the outside, the IP value is less. The royalty savings might be valued at 8% [ReillyS:98].

When future income is based on many factors, a fair allocation of future income to software using the Pareto rule, assuming an equal benefit for software investments, can easily be false. At times, software investments can have disproportionate benefits, for instance when an industry segment is moving from traditional workflow to modern strategies. An allocation based on cost data assumes broad management competence; unfortunately, in companies where management is not software savvy, their spending on software is often not rational. Sometimes ambitious internal efforts are undertaken, and their development costs become excessive, so that a cost-based allocation will exaggerate the value of software. In those cases, using estimates of what the software should have cost can provide a more rational balance. Those can be computed using function points, and, since the software exists, that task is simplified [Jones:98]. The estimate of what the cost should have been can only be used as the nominator in the Pareto optimum equation that allocates the total revenue, not as surrogate for its value in the future.

6.4 Future Allocation

For the valuation, the current SW 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. Our sample illustration below allocates 25% of the net revenue to the software IP component, based on Case 1 of Section 6.3, a well-managed company which develops and sells software.

To arrive at today's value of that profit, the benefit of future sales has to be discounted into the future [ReillyS:98]. We use here a typical software industry 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 of average risk and use a beta of 1.0.

7. 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 to today. Finally we can sum up the contributions from each future year in the horizon.

7.1 Computing the Value of the Software

A spreadsheet provides the best means to summarize the evaluation [Wiederhold:05]. The simple general formulas used by economists, especially if they assume perpetual benefits, and fixed cost/revenue ratios, do not allow for the software engineering factors we consider here [GordonG:97]. 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 to 9 years. The annual IP contribution is that of the first version, the initial software being valued, using the reduction shown in Figure 3.

Factors \ Year	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**-1*	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 1- ***Rep***	1.00	0.95	0.90	0.85	0.80	0.75	0.70	0.65	0.60	0.55
Fraction old ***Ini/R***	100%	57%	39%	28%	22%	17%	14%	11%	9%	8%
Unit **p**rice ($) (rule P)	500	500	500	500	500	500	500	500	500	500
Units **s**old (Fig.4)	0	1911	7569	11306	11395	8644	5291	2740	1241	503
Revenue (K$) ***Us×Up***	0	956	3,785	5,653	5,698	4,322	2,646	1,370	621	252
SW **s**hare @ 25%×***Rev***	0	239	946	1,413	1,424	1,081	661	343	155	63
From **i**nitial SW ***F×Ss***	0	136	365	400	311	187	93	39	14	5
Discount @15% ***1/1.15^y***	1.00	0.87	0.76	0.66	0.57	0.50	0.43	0.38	0.33	0.28
Discounted **s**hare ***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$								

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

The result shows that selling a total of ***ΣUs*** ≈ 50 000 units of a software product at a price of ***Up*** = $500 over 9 years yields a bit less than a million dollars (***ΣIP*** = $0.989M) in terms of IP value discounted to the date of placing the initial software into service. The company also increased in value due to its ongoing investments in software development from the 25% share of revenue allotted to software, although much of that will be needed for maintenance. Furthermore, the company also provides jobs and societal benefits.

7.2 Discussion

Note that the revenue ***Rev*** was much greater than the value of the IP realized in the end. The IP value can only be realized by spending on ongoing SW development, marketing, and all the other items within a business. The ***Ss*** = 25% share of revenue allocated to the SW contribution amounted to about $3.7M is available for ongoing software development and growth. Much of that will be spent on maintenance cost, as be shown in Table 2.

We have 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. For instance, in practice, version frequency reduces as software gets more mature. It is also possible to plug multiple alternative 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 will look much better if the unit price ***Up*** is set to $2000. However such a change means that many individuals will be loath to buy the software, salesmen will have to work harder, and a purchaser will have to deal with committees. All that bother will decrease sales and delay the generation of income.

7.3 Mature Software

The valuation shown here was performed for new software. When the software has been in existence for some time, the same method can be used, but the parameters must be adjusted. While the remaining life will be less, a diminution curve, as shown in Figure 3, is now applied to a later version, and will be less steep. The sales figures become more predictable as well. Furthermore, once a product has survived the initial introduction, the risk will be less, and discount rate can be lower.

But maintenance of the software will still be needed, and an assumption sometimes made, that the creator can just sit back and collect income, is unrealistic.

7.4 The Accountant's View

The simplified approach above provides an adequate estimate for planning software development. But if you need to call in accountants and economists to formally justify a project they will want to be more precise. First of all, rather than assuming annual receipts at the end, they would typically break the years down in quarters or months, and then assume that the income arrives in the middle of the periods -- the mid-month convention. That can add a few percent to the valuation. But then, one must consider the delay the business allows for the customer to pay for the product, and that will delay the income and hence reduce the value.

In Section 8.3, where we also consider the lag between spending on software maintenance and deriving profit from it, similar refinements will be made. In our example the lag between expenditure and profit is set to be two years, as the total of development and adoption delays. Development delays will be less if change was planned for in the software architecture [KruchtenOS:06]. Again, quarterly or monthly accounting of expenses will refine the estimates. Adoption delays will differ depending on the business model. It should be little if software updates are included in the license or if the software is used internally. If marketing is needed to convince purchasers, then the delay will be greater. In the end, given the uncertainty of determining the future, following accounting principles does not make a difference when these predictions are used to support decision making.

8. A Service-oriented Business Model

We indicated in Section 3 that maintenance has a high benefit. A service-oriented business model exploits the value generated by maintenance. In this approach the relationship of supplier to customer is maintained after the initial sale. Using our model for a quick investigation shows that value. 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 in Section 7.1, since it depends on efforts made in future years. In the conclusion we will comment on the operational aspects of such a service-oriented business model.

Two new financial streams can 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*. We now spend much of the software share shown in Table 1.

Income from maintenance changes the cashflow drastically. After year 6 the income of maintenance exceeds the income from sales. Extending Table 2 beyond year 9 we find that by year 11 software sales are negligible, but maintenance revenues remain substantial. Figure 6 plots the effect as a dashed line, all amounts are in thousands. But to generate those revenues maintenance of the product is required, which eventually becomes costly and exceeds the total income, limiting the economic lifetime of the software product for the supplier.

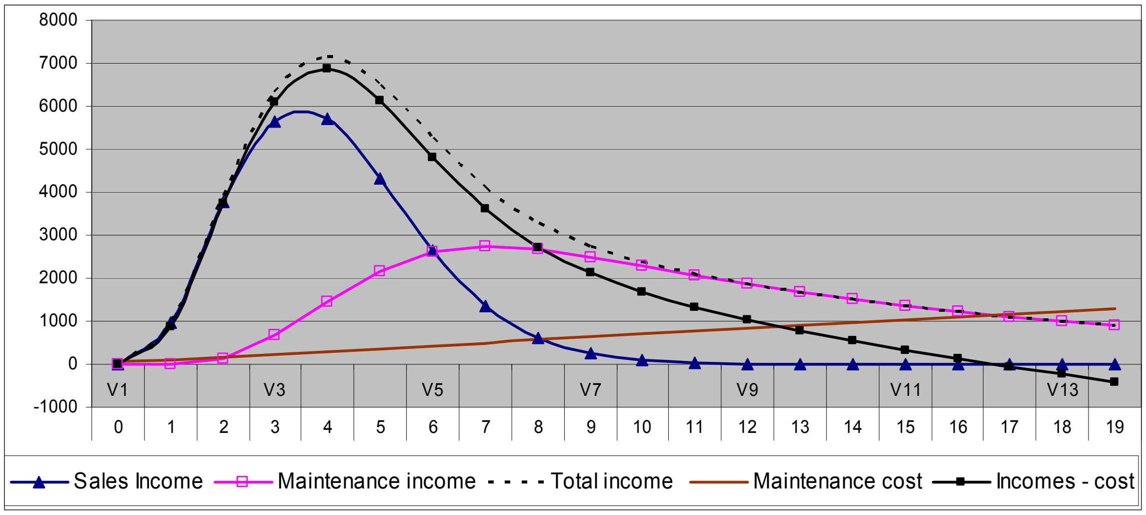

Figure 6. Maintenance income versus sales income, and the total after cost

8.1 Income from Maintenance.

A supplier of enterprise software obtains an ongoing benefit by collecting annual maintenance fees. The software earns maintenance income every year, even as sales are made to new customers. Most customers renew their contracts since the alternative, obtaining new software, is much more costly than paying a maintenance fee. A seller of shrink-wrapped software has to provide enough incentives to make customers purchase new versions. Table 2 shows the income for 15% annual maintenance fees, based on the initial price of enterprise software. Assumptions are that first year maintenance is free and that 90% of customers renew their maintenance contract annually. That income is available for the next year. Not all of the income from maintenance fees is available for maintaining software. For instance, there has to be a better help desk. For our example 2/3 of the fee is made available for software tasks, as estimated in Section 5.2.

8.2 Cost of Maintenance.

We also have to consider software maintenance costs throughout our horizon. Maintenance costs are best estimated based on initial software cost. For our example we make the assumption here that the initial software cost was $500K, about half of the computed value obtained in Table 1, leaving the other half to other initial costs. Spending much more would be unwise given its discounted value, and spending much less for marketable and maintainable software is unlikely. 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, and this is accounted for by making the cost proportional to the total, ever growing, code size, as shown in Figure 8. As a cross check we compute the actual maintenance cost from year 2 to year 7, and find that the cost is $2,075K. In this case long-term maintenance consumes about 80% of the total cost, a typical fraction for software having that lifetime [McKinsey:03].

8.3 Lag or Gestation Period

When computing the IP we were not concerned about costs. But when we consider costs now we also have to be concerned about when the costs were incurred. Lag, sometimes referred to as the gestation period, is the delay between the time of the investment and the time that the investment brings in income. There are two components of lag: the development lag, incurred prior to the release of the product, and the market lag, after the release of the product. The sale curves shown in Section 5.2 already dealt with the market lag, so that now we must still consider the development lag for initial product introduction.

The total amount of effort spent grows steadily during the development period. The ratios of effort spent on research, development, and testing will differ from product to product. Figure 6 sketches the efforts involved in releasing a new product to its market.

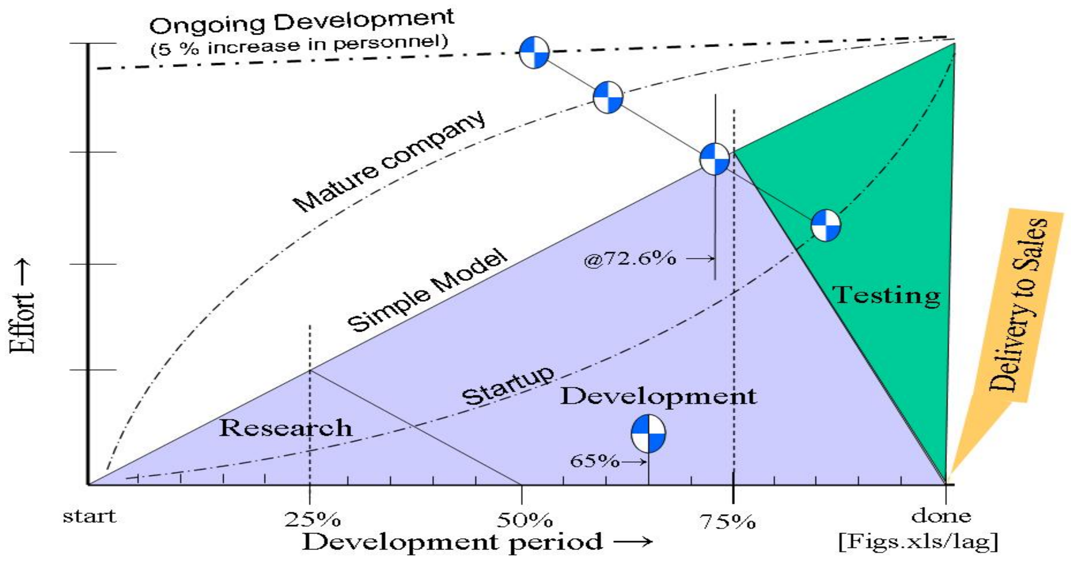

Figure 7. Effort distribution prior to release of a new product.

Given a simple model, as shown in Figure 7, for the components and timings of the development efforts, we can estimate the efforts and when they take place. The reduction of development effort when testing starts allows the development teams to work on successor products in the same product line and to start devising new product lines when capabilities are needed that are not compatible with the architecture in the current product line. When most of the development is for maintenance, then assuring that prior functionality continues requires extensive regression testing, and the testing team will be larger than the development team [Maraia:05].

A mature company will be able to ramp a development effort more rapidly. Such a personnel allocation would reduce the total development time, but increase the percentage distribution used to compute the effective lag. On the other hand, a startup company may have to work initially with very few resources, and only after it demonstrates feasibility can attract sufficient venture capital to move towards product completion. It may spend half of its investment in the last 3 months of a two-year development period, and its effective lag time will be equal to three months. We indicate those choices by dot-dashed lines in Figure 7. Within those parameters the total period required for the development is determined by the size and complexity of the product.

If only maintenance, as defined in Section 3, is performed then the total effort expended can be flat. Research on items to be adapted or perfected can extend to late in the development interval, and testing can start early. Small changes can be immediately submitted to testing, since the base software exists already [Cusumano:95].

Predicting effective lags. For the typical case we can compute the centroids for the efforts, allowing a single point to replace the actual aggregation in our estimations [Wiederhold:07]. The results for the simple model are

1. The total effort has a centroid at 33% before product release
2. Research: 12.5% of total effort, centered at 75% before product release
3. Testing: 25% of total effort, centered at 8.3%% before product release

4. Combined development and testing: 87.5% of total effort, centered at 27.4% before product release
5. Actual development: 62.5% of total effort, centered at 35% before product release.

The centroids computed here ignore any discounting to the NPV at the actual release date of the software. Early expenses not compensated by income accumulate interest charges or their equivalent in investor expectations. Discounting to the time of delivery increases early costs and moves the centroids to an earlier time.

New products also have substantial research risks, as well as unknown competitive threats requiring adaptation and flexibility. Minimizing lag is crucial, but simplified by not having to cater to an existing community. For subsequent maintenance versions less research will be needed, so that the lag patterns differ for new products versus successor versions. No new staff may be needed, so that the total effort can be level. For substituting versions the lag depends on the extent and interaction of updates required. As product versions become more complex, lag is likely to increase, unless changes to successor versions are reduced. Maintenance lag is largely due to the need of testing the interaction of new code with all remaining code in a version. If changes become too great, it becomes impossible to have a reliable successor version [Bernstein:03].

Substitution products, completely new products that must cater to an existing customer base incur the most lag, and corresponding risks. It is impossible to know in full detail how all customer use the prior product. Customers will be justifiably wary of such products. Some companies have gone out of business when substitution products created excessive errors. Sticking with upgraded successor versions is much more common.

The maintenance costs for successor versions are borne by the income from ongoing software and maintenance licenses sales. But those costs overlap the costs needed to keep the software product viable for sales. Since the benefits of maintenance expense are delayed, but paid for with current funds, its contribution must be discounted -- we must also compute a total lag here. The date that a software version is released falls between those two times. Given a development time of one year for maintenance code, testing for a half a year, and new versions being released to the market every 18 months the average lag is assumed here to be at 1.5 years.

This delay is sufficiently long so that a discount rate should be applied, since the costs precede the income by 1.5 years. Here imposition of a discount rate increases the cost relative to the time when income is received. The effect is that effective maintenance costs are increased by a steady 23% to account for the 15% annual discount, raising the accounting charges of maintenance. Still, by the third year the allocated income from maintenance fees is sufficient to pay for all maintenance.

8.4 Combining it all in a service-oriented model

By considering maintenance costs and revenue as well as sales income the summary in Table 2 can only show income, not initial IP value. Even with the high cost of long-term ongoing maintenance, with the maintenance income the actual cash flow increases nearly three times over the 9-year time-frame. However, since much these benefits will occur in later years, the income, when discounted to today, increases less. The discounted income is now \$7.35M rather than \$3.69M, still about twice as much as without the income due to maintenance.

Additional factors in a service-oriented model	Year									
	Today	y1	y2	y3	y4	y5	y6	y7	y8	y9
Est. initial SW cost ($K)	500									
Cost of maintenance ***Cm***	67	100	167	233	300	367	433	500	567	633
discounted for lag ***Cl***	82	123	206	288	370	452	534	617	699	781
SW ***r***etained **Σ*Rev*×0.9** ^{y}	n.a.	860	4,559	9,756	14,478	17,352	18,262	17,806	16,646	15,233
Revenue ***l***eft ***Rev - Cl***	-82	832	3,579	5,365	5,328	3,870	2,111	753	-79	-529
Maint.***r***evenue ***Sr***$_{-1}$×***0.15***	none	none	129	684	1,463	2,172	2,603	2,739	2,671	2,497
Maint.revenue ***a***vail ***2/3***	none	none	86	456	976	1,448	1,735	1.836	1,781	1,665
Discounted M.rev. ***Mr/D***	none	none	98	450	837	1,080	1,125	1,030	837	710
SW share ***l***eft @ 25%×***Rl***	-21	208	981	1,797	2,307	2,415	2,263	2.015	1,761	1,532
Discounted to ***t***oday ***Sl*×*D***	-21	181	742	1,182	1,320	1,201	979	758	576	436
9 years disc. total ***ΣDt***	7,350	K$								

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

More income will accrue beyond year 9, but the amount becomes small, less than 10% of the sales in the best year, year 4. Table 2 shows that soon after year 7 the cost of maintenance exceeds the income from sales, so that without the maintenance income it would be wise to stop improving the product at that time. But maintenance income has become as great as sales income at that time, so we can profitable continue the service business after year 7. But all good things must end, and after year 15 total revenue, mainly sales of maintenance licenses, is less than the ever increasing maintenance costs.

Stopping abruptly generates bad press for a company that sells other products as well. Gradually reducing maintenance as less income is generated is a wiser option. Still, when customers find out that maintenance is reduced, sales will diminish rapidly and fewer maintenance licenses will be renewed.

To assess the effect of gradual reduction of maintenance after the last profitable version (*v*11.0), we recomputed the estimate assuming no more sales after year 16 and a maintenance customer retention of only 60% annually. The maintenance reductions actually lead to some temporary net gain. But by year 16 maintenance income no longer covers even the reduced maintenance cost and the service is withdrawn. Figure 8 includes the effect in year 17 and beyond. The additional income gained is nearly $3.5M, but when discounted to today its value is only $0.735M [Wiederhold:05]. Losses would be further reduced by spending less on marketing, but that decision goes beyond our software model.

Plotting the data allows rapid review of alternatives. For instance making initial sales often has a high sales engineering expense, say 40%, while selling and installing a new maintenance version has a much lower expense, say 20%. In Figure 8 we also show the resulting adjusted curves, as well as the ever increasing maintenance cost, shown as a deduction.

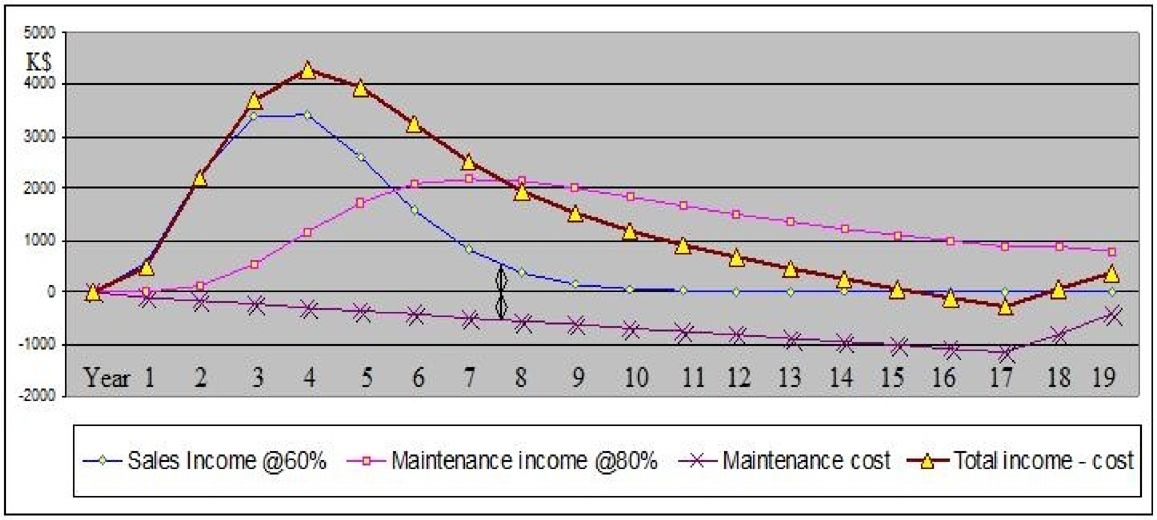

Figure 8. Maintenance, sales income, and cost after an expense adjustment

8.5 Model differences.

Obtaining a result for a service-oriented business model required many more assumptions than the base IP valuation for the initial IP value of software, as shown in the reasoning leading to Section 6. We had to consider ongoing costs as well as income. We also had to make a rough guess at the initial cost, an issue normally reserved for return-on-investment (RoI) estimates. RoI calculations rarely include maintenance costs, but will include the cost of initial capital and other financial concerns, which we are able to neglect. Section 7 does quantify the value of retaining customers, a point made recently by [Larcker:04]. Operating in an effective service model also requires changes to business practices, with costs that are hard to quantify. For sellers of shrink-wrapped software additional investment will be needed to convince customers to buy the latest version. When software is developed for internal use it is hard even to track the customers and their use of the software [TillquistR:05]. In the model we used such detail is ignored, the focus was on aggregate projections.

The beneficial difference of including maintenance efforts is not captured in a traditional IP valuation. The renewal by maintenance licenses means that new programs being sold and those maintained at the customer site are identical. But modeling an enterprise business in this manner obeys the appraisal rule that one should value a business as it actually operates, not use a model that builds one version of software and then exploits it until it has no value [AP:04].

9. Conclusions

We have a primary conclusion, directly relevant to the topic, but in the process of analysis also recognized some aspects of software life, engineering processes, management, and education that are worthwhile to mention here.

9.1 Valuation of software is possible.

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 had to place it into an overall business model. This model allows a quantitative assessment if the value of the inherent intellectual property, using information commonly needed for IP assessments. Without such a model amortization schedules have been used, that typically bear little relationship to the processes that represent maintained software. Still, many assumptions must be made 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 done in Section 7, is difficult if we can't even tell what the value of software is in the first place.

9.2 Maintenance of software is worthwhile.

A specific conclusion is that maintaining software, although costly, is very worthwhile. Maintenance provides the required continuing refreshment of the inherent IP. My opinion is that ongoing costs for maintenance should not be accounted for as R&D expenses, but rather as cost-of-goods sold. Such a change would means that the gross-margin financial indicator could become meaningful, as discussed in Section 2.3, Making the distinction would also provide insight on what a company is actually spending on innovative R&D and on keeping its product viable in the marketplace. I do realize that the accounting profession is not ready for such a change. Even the current FASB rules on software capitalization after feasibility of the product is established are largely ignored [Lev:04].

With high quality maintenance and income from that maintenance a company transforms its business from a sales model into a service model. 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. Such maintenance provides the continuous improvement or incremental innovation that complements the less frequent disruptive innovations that are part of the lifecycle of successful organizations [ClaytonR:03]. But operating in a service model also means that management must appreciate the process and software staff must be motivated to sustain the value of the product by performing excellent maintenance [LandsbaumG:85]. Not only are the programmers involved in ongoing improvements of the software, but also the sales and marketing staff who provide the ongoing intellectual input for the three types of maintenance presented in Section 3.2.

9.3 Maintenance costs limit the life of software.

However, eventually maintenance costs overtake income, as shown in Section 7. The complexity of growing software does incur high costs [BeladyL:72]. Providing for ease of maintenance in software design and managing maintenance well should have significant benefits. Still, the limit on the life of software under maintenance is not obsolescence, but rather the cost of maintenance, as recognized many years ago [Datamation:68]:

Program complexity grows until it exceeds the capacity of the programmer to maintain it.

This observation answers the vexing question: How can software wear out? While maintenance contributes to long life, the growth it engenders in the end makes that maintenance too costly. A survey of software lifetimes showed 11.2 years for systems substitution because of maintenance costs, other issues were hardware replacements and switching to on-line operations, reducing the average lifetime to 8.8 years [TamaiT:92]. For a business to survive in the long term, maintenance and innovation must be balanced.

9.4 Education ignores the major cost component of software.

Education and technological attitudes favor novelty over maintenance. A typical and popular software engineering textbook devotes 3 out of 850 pages to maintainability and maintenance, although that same book states that the effort devoted to maintenance is greater than 60% of total cost [Pressman:01]. Today most students in computing disciplines graduate without ever having faced the issue that software must be maintained to adapt to changing situations. 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 of maintenance, it mainly reduces the benefits of maintenance. Managers often bemoan the high cost of maintenance in their organizations, since they are not clear about the benefits [Spolsky:04]. In such settings experienced personnel is reassigned to new tasks that seem more exciting [Pfleeger:01]. In a company where management understands software economics and its leverages those interns and recent graduates will do innovative work, and the experienced staff who sustain the value of their existing products and services will be well rewarded. This process has been referred to as a "gentrification" of maintenance [Foote:98].

9.5 Problems not addressed.

A similar analysis is likely to be appropriate for content databases, which have become a recent focus for intellectual property protection. Those databases are also subject to continuing maintenance and improvements [GardnerR:98]. However, I have not had access to case data to validate content-based models.

The analysis approach shown here cannot deal with the valuation of open-source software; current information on its use is scant [SamoladasSAO:04]. The economic benefits of open-source software are hard to measure. We know that open source software also requires maintenance to remain effective [Spolsky:02]. Companies that tried to create a business from servicing open-source software have had a very hard time, since reasonable maintenance costs seem extremely high when software acquisition costs are nearly negligible. An interesting analysis shows that under certain conditions the acquisition value of software is less than 20% of the value of maintenance and upgrade warrants [Lefkowitz:05].

9.6 Benefits of valuation analysis.

Even if though the analyses to value software depend on assumptions that cannot be verified in advance, having the assumptions stated explicitly allows a discussion of design and business alternatives. Opinions and disagreements are brought down to a less emotional level, and will include more organizational and technical specifics. Having a quantified valuation method that generates results to support such discussions is a necessary tool.

Acknowledgements.

Discussions with Treasury economists, as Charles Adelberg and Joy Yen helped in establishing the principles shown here. A course directed by Jeremy Dent at the London Business School helped in understanding valuation issues in general. Laurence Melloul provided references for information regarding Function Points. I received constructive feedback from Avron Barr, Dan Grosshans, Steve Hoffer, Shirley Tessler, Bhavani Thuraisingham, David Waxman, John Wiederhold, and the ACM reviewers. Discussion with colleagues and students at the University of Arizona Eller School of Business provided further focus. Any errors in this paper are of course, my responsibility, but I will not assume the responsibility for business decisions made based on use of the presented model.

Biography.

Gio Wiederhold is now an emeritus professor of Computer Science, Electrical Engineering, and Medicine at Stanford University. Born in Italy, and raised in Germany and the Netherlands, he spent 16 years in industry. In 1976 he obtained a PhD in Medical Information Science from the University of California, San Francisco and joined the faculty at Stanford University. He is a fellow of the ACM, the IEEE, and the ACMI. His web page is at http://infolab.stanford.edu/people/gio.html.

References.

Most of the references in this sections are cited, but some, marked †, while perused, were not cited.

[Abba:97] Wayne Abba: "Earned Value Management: Reconciling Government and Commercial Practices"; *Program Manager,* Vol.26, pp. 58–69.

[AP:04] The Appraisal Foundation: *Uniform Standards of Professional Appraisal Practice;* Standard 9, 2004 Edition.

[Aron:83] J.D. Aron: *The System Development Process, The Programming Team (Part II)*; Addison-Wesley, 1983.

[BankerS:97] † Rajiv Banker and Sandra Slaughter: "A Field Study of Scale Economics in Software Maintenance"; *Management Science*, Vol.43 no.12, December 1997, pp.1709-1725.

[BankerDS:97] † Rajiv Banker, Gordon Davis, and Sandra Slaughter: "Software development Practice, Software Complexity, and Software Maintenance Performance: a Field Study "; *Management Science*, Vol.44 no.4, April 1998, pp.433-450.

[BaruaM:89] † A. Barua and T. Mukhopadhyay: "A Cost Analysis of the Software Dilemma: To Maintain or to Replace"; Proc. of the 22nd Hawaii Internat. Conf. on System Science, 1989, pp.89-98.

[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.

[Batelle:73] Batelle Laboratories: *Evaluation of the Implementation of a Medical Information System in a General Community Hospital*; report on a study commissioned by NCHSRD, DHS, 1973.

[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.

[BennigaS:97] Simon Z. Benning and Oded H. Sarig: *Corporate Finance: A Valuation Approach;* McGrawHill, 1997.

[Bernstein:03] Philip Bernstein: Remark at NLM/NSF planning meeting, Bethesda, MD, 3 Feb 2003.

[Blum:98] Bruce I. Blum: Closing the circle; in [Glass:98], pp.238-249.

[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.

[BranscombEA:91] Lewis Branscomb (chair) et al.: Intellectual Property Issues in Software; Computer Science and Telecommunications Board, National Research Council, National Academy Press, 1991.

[Brooks:95] Frederick Brooks: *The Mythical Man-Month, Essays in Software Engineering;* Addison-Wesley, 1975, reprinted 1995.

[BW:05] Business Week: Global Brand Scoreboard, the Top 100 Brands; *Business Week*, 1 August 2005, pp. 90-93.

[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.

[ChoiSW:97] † Soon-Yong Choi, Dale O. Stahl, and Andrew B. Whinston: *The Economics of Electronic Commerce;* Macmillan, 1997.

[ClaytonR:03] Clayton M. Christensen and Michael E. Raynor: *The Innovator's Solution: Creating and Sustaining Successful Growth; Harvard Business school, 2003.*

[CorradoS:04] Carrol Corrado and Dan Sichel: "Measuring Capital and Technology: An Expanded Framework"; Federal Reserve, August 2004, http://www.federalreserve.gov/pibs.feds/2004/200465/200465pap.pdf.

[Panetta:89] Karen Panetta: Observation on systems implementation; DEC, 1989.

[CockeM:90] † John Cocke and Victoria Markstein: "The Evolution of RISC Technology at IBM; *IBM Journal of Research and Development,* Vol.44 No.1/2, January/March 1990, pp.48-55.

[Cusumano:95] Michael Cusumano and Richard Shelby: *Microsoft Secrets, How the World's most Powerful Software company Creates Technology, Shapes Markets, and Manages People*; MIT, 1995.

[Cusumano:04] Michael A. Cusumano: *The Business of Software;* Free Press, 1998.

[Damodaran:2002] Aswath Damadoran: *The Dark Side of Valuation: Valuing Old Tech, New Tech, and New Economy Companies;* Prentice-Hall, 2002.

[Datamation:68] "Program complexity grows until it exceeds the capacity of the programmer to maintain it."

[EcclesEA:01] † Robert G. Eccles, Robert H. Herz, E. Mary Keegan, David M. H. Phillips: *The Value Reporting Revolution: Moving Beyond the Earnings Game*; Price-Waterhouse-Coopers, 2001

[Economist:06] The Economist: "Getting a Grip on Prosperity"; The Economist, 4 March 2006, p.72.

[FeigenbaumMNP:88] Edward Feigenbaum, Pamela McCorduck, H.Penny Nii, and Tom Peters: *The Rise of an Expert Company: How Visionary Companies Are Using Artifical Intelligence to Achieve Higher Productivity and Profits*; Macmillan, 1988,

[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.

[Gates:98] Bill Gates: "Compete, don't delete"; *Economist*, 11 June 1998.

[Gates:04] Bill Gates: "Losses due to copying must be balanced with disincentives and costs of protection methods"; quote during a discussion on "Building Confidence in a Connected Marketplace", 1 Oct 2004, Computer History Museum, Mountain View, CA.

[GarmusH:01] David Garmus and David Herron: *Function Point Analysis: Measurement Practices for Successful Software Projects*; Addison-Wesley Information Technology Series, 2001.

[Gilb:05] Tom Gilb: *Competitive Engineering*; Elsevier, 2005.

[Glass:98] Robert L. Glass: *In the Beginning: Recollections of Software Pioneers;* IEEE Press, 1998.

[Glass:03] Robert L. Glass: *Facts and Fallacies of Software Engineering*; Addison Wesley, 2003.

[Gordon:82] Myron J. Gordon: *The Investment, Financing and Valuation of the Corporation*; Irwin, 1962; reprint Greenwood Press, 1982.

[GordonG:97] Joseph R. Gordon and Myron J. Gordon: "Finite Horizon Expected Growth Model"; *Financial Analysts Journal,* Vol.53, No.3, May June 1997, pp.52-61.

[Hall:99] Robert E. Hall): "The Stock Market and Captial Accumulation"; NBER Working Paper 7180, National Bureau of Economic Research, Cambridge MA, June 1999.

[HandL:03] John Hand and Baruch Lev (editors): *Intangible Assets: Values, Measures. and Risks*; Oxford University Press, 2003.

[Hatton:05] Les Hatton: Estimating source lines of code from object code; CISM, Kingston Ontario, Aug.2005, http://www.leshatton.org/Documents/LOC2005.pdf.

[Hendler:02] James Hendler et al.: Report on Database Migration for Command and Control; United States Air Force Scientific Advisory Board, SAB-TR-01-03, Nov. 2002.

[HennessyP:90] John Hennessy and David Patterson: *Computer Architecture*; Morgan Kaufman, 1990.

[Heylighen:97] Francis Heylighen: Occam's razor; Principia Cybernetica Web, 1997.

[Hulten:06] Charles Hulten: "Intangible Capital and Economic Growth"; NBER Working Paper 11948, Jan.2006, http://papers.nber.org/tmp/16760-w11948.pdf

[IEEE:98] IEEE Standard for Software Maintenance; Document 1219-1998, IEEE, 1998.

[Jones:95] Capers Jones: "Backfiring: Converting Lines of Code to Function Points"; *IEEE Computer,* Vol.28, No.11, November 1995, pp. 87-88.

[Jones:96] Capers Jones: *Applied Software Measurement: Assuring Productivity and Quality*; McGraw-Hill, 1996.

[Jones:98] T. Capers Jones: *Estimating Software Costs*; McGraw-Hill, 1998.

[Kemerer:93] Chris Kemerer: "Reliability of Function Points Measurement: A Field Experiment"; *Comm. ACM,* Vol. 36*, No. 2,* February 1993, pp. 85-97.

[KruchtenOS:06] P. Kruchten, H. Obbink, J. Stafford: "The Past, Present, and Future for Software Architecture"**;** *IEEE Software*, v. 23, no. 2, pp. 22- 30.

[LandsbaumG:85] Jerome B. Landsbaum and Robert L. Glass: *Measuring and Motivating Maintenance Programmers;* Prentice-Hall, 1985.

[Larcker:04] David Larcker: in "Back to the Drawing Board: Is the Traditional Theory of the Firm Obsolete?"; *Strategic Management,* Wharton On-line report 1047, Sep. 2004 <http://knowledge.wharton.upenn.edu/article/1047.cfm>

[Lefkowitz:05] Robert Lefkowitz: Calculating the True Price of Software; *O'Reilly OnLamp*.com, 21July2005.

[LeonardS:05] Gregory Leonard and Lauren Stiroh (editors): Economic Approaches to Intellectual Property Policy, Litigation, and Management; National Economic Research Associates (NERA), White Plains, NY, 2005.

[Lev:01] Baruch Lev: *Intangibles, Management, Measurement and Reporting;* Brookings Institution Press, 2001.

[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.

[Maraia:05] Vincent Maraia: *The Build Master : Microsoft's Software Configuration Management Best Practices*; Addison-Wesley Microsoft Technology, 2005.

[Marciniak:94] John J. Marcianak: *Encyclopedia of Software Engineering*; Wiley, 1994.

[McGraw:03] Gary McGraw , "From the Ground Up: The DIMACS Software Security Workshop"; *IEEE Security & Privacy* **1** (2), March/April 2003, pp. 59-66.

[McKinsey:03] McKinsey: "Fighting Complexity in IT"; *McKinsey Quarterly,* April 2003.

[Mookerjee:05] Radha Mookerjee: "Maintaining Enterprise Software Applications"; Comm. ACM, Vol.48, No.11, Nov. 2005.

[Moore:95] Geoffrey A. Moore: *Inside the Tornado: Marketing Strategies from Silicon Valley's Cutting Edge*; Harper Business, 1995.

[MurphyL:00] Brendan Murphy and Bjorn Levidow: Windows 2000 Dependability; to appear in *Proc IEEE International Conference on Dependable Systems and Networks*, June 2000; Microsoft Technical report MSR-TR-2000-56.

[NodderWD:99] Chris Nodder, Gayna Williams, and Deborah Dubrow: "Evaluating the usability of an evolving collaborative product —changes in user type, tasks and evaluation methods over time"; *Proceedings of the international ACM SIGGROUP conference on Supporting group work*, Phoenix, Arizona, United States, 1999, pp150 - 159

[Parnas:94] David Parnas: "Software Aging"; 16th Conf. on Software Engineering, May 1994, pp. 279-287.

[Park:92] Robert E. Park: "Software Size Measurement: A Framework for Counting Source Statements"; Software Engineering Institute, *Technical Report CMU/SEI-92-TR-20*.

[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.

[Polk:06] Michael Polk: "It's all about Dislocationg Ideas"; *Knowledge@Wharton*, 15 Nov 2006, http://knowledge.wharton.upenn.edu/article/1604.cfm .

[Porter:98] Michael E. Porter: *Competitive Strategy*; The Free Press, Simon and Schuster, 1998.

[Pressman:01] Roger Pressman: *Software Engineering, A Practioner's Approach*; 5th edition; McGrawHill, 2001.

[Razgaitis:03] † Richard Razgaitis: *Valuation and Pricing of Technology-Based Intellectual Property*; Wiley, 2003

[Rechtman:01] Ygal Rechtman: Accounting Treatment of Intangible assets; draft, http://www.rechtman.com/acc692.htm, July 2001.

[ReillyS:98] Robert F. Reilly, and Robert P. Schweihs: *Valuing Intangible Assets;* Irwin Library of Investment and Finance; McGraw-Hill, 1998.

[Rottman:06] Joseph P. Rottman: "Successfully Outsourcing Embedded Software Development"; *IEEE Computer*; Vol. 39, No. 1, January 2006, pp. 55-61.

[Roux:97] David Roux: Statement and sketch about software growth; 1997.

[Samuelson:83] Paul A. Samuelson: *Foundations of Economic Analysis*; Harvard University Press, 1947, 1983.

[SamoladasSAO:04] Ioannis Samoladas. Ioannis Stamelos, Lefteris Angelis, and Apostolos Oikonomu: "Open Software Development Should Strive for Even Greater Code Maintainability"; *Comm. ACM*, Vol.47 No. 10, Oct. 2004, pp.83-87.

[SiewiorekBN:92] † Daniel P. Siewiorek, C. Gordon Bell, Allen Newell: Computer Structures, Principles and Examples; McGrawHill, 1982.

[SmithP:00] Gordon Smith and Russell Parr: *Valuation of Intellectual Property and Intangible Assets*, 3rd edition; Wiley 2000.

[Spolsky:02] Joel Spolsky: The Economics of Open Source; 12 June 2002, www.**joel**onsoftware.com/articles/StrategyLetterV.html, reprinted in [Spolsky:04]

[Spolsky:04] Joel Spolsky: *Joel on Software*; Apress, 2004.

[Stein:74] Herbert Stein, chairman of the Council of Economic Advisors, Nixon administration: quote on extrapolation of trends; 1974.

[Stobbs:00] Gregory Stobbs: *Software Patents*; Aspen publishers, 2000.

[Takahashi:05] † ShigeruTakahashi: The Rise and Fall of Plug-Compatible Mainframes; *IEEE Annals of the History of Computing*, January-Match 2005.

[TamaiT:92] Tetsuo Tamai and Yohsuke Torimitsu: "Software Lifetime and its Evolution Process over Generations"; Proc. Conf. on Software Maintenance, *ICSM*, Nov. 1992, pp.33-37.

[Tamai:02] Tetsuo Tamai: "Process of Software Evolution"; *First International Symposium on Cyberworlds*, Tokyo, Nov. 2002, p.8-15.

[TanM:05] † Y. Tan and V.S. Mookerjee "Comparing uniform and flexible policies for software maintenance and replacement"; IEEE Trans. on Software Engineering, Vol. 31, No. 3, March 2005, pp. 238 - 255.

[TillquistR:05] John Tillquist and Waymond Rodgers: "Using Asset Specificity and Asset Scope to Measure the Value of IT"; *Comm. ACM,* Vol.48 No. 1, Jan. 2005, pp. 75:80.

[Wheeler:05] David A. Wheeler: SLOCCount; http://www.dwheeler.com/sloccount/, 2005.

[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://infolab.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:04] Gio Wiederhold : "Outsourcing as Export of Intellectual Property"; KD (Knowledge Discovery) Nuggets, 10 March 2004, http://www.kdnuggets.com/news/2004/n05/3i.html.

[Wiederhold:05] Gio Wiederhold: Simple spreadsheets for the tables and graphs shown here and in [Wiederhold:06]; links at http://infolab.stanford.edu/pub/gio/inprogress.html#worth, 2005.

[Wiederhold:06] Gio Wiederhold: "What is your Software Worth?"; *Comm. ACM*, Vol.49 No.9, September 2006, p.65-75.

[WiederholdGM:06] Gio Wiederhold, Amar Gupta, and Rajat Mittal: The Value of Outsourced Software; Eller College of Management Working Paper No. 1031-06, University of Arizona - Eller College of Business and Public Administration, 14 May 2006; available at the Social Science Research Network, http://ssrn.com/abstract=905148.

[Wiederhold:07] Gio Wiederhold: Determining Lag; manuscript in preparation, 2007.

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