Industrial and Economic Properties of Software
Technology, Processes, and Value

David G. Messerschmitt
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California, USA
messer@eecs.berkeley.edu

Clemens Szyperski
Microsoft Research
Redmond, Washington, USA
cszypers@microsoft.com

Copyright notice and disclaimer

Abstract

Software technology and its related activities are examined from an industrial and economic perspective. More specifically, the distinct characteristics of software from the perspective of the end-user, the software engineer, the operational manager, the intellectual property lawyer, the owner, and the economist are identified. The overlaps and relationships among these perspectives are discussed, organized around three primary issues: technology, processes, and value relationships. Examples of the specific issues identified are licensing vs. service provider models, alternative terms and conditions of licensing, distinct roles in the supplier value chain (development, provisioning, operation, and use) and requirements value chain (user needs and requirements), and the relationship of these issues to industrial organization and pricing. The characteristics of software as an economic good and how they differ from material and information goods are emphasized, along with how these characteristics affect commercial relationships and industrial organization. A primary goal of this paper is to stimulate more and better research relevant to the software industry in the economic, business, and legal disciplines.

Table of contents

1 Introduction

The software industry has become critical. It is large and rapidly growing in its own right, and its secondary impact on the remainder of the economy is disproportionate. In view of this, the paucity of research into the industrial and economics properties of software—which flies in the face of both its growing economic importance and the interesting and challenging issues it engenders—is puzzling. Most prior work on the economics of software—performed by practitioners of software engineering—has focused on serving software developers, where the emphasis is on cost estimation and justification of investments, and to a lesser extent, estimation of demand [Boe81, Gul93, Ver91, Boe99, Boe00, Ken98, Lev87, Cla93, Kan89, Boe84, The84]. As discussed later, many of the concepts of information economics [Sha99], such as network externalities [Kat85], lock-in, and standardization [Dav90], also apply directly to software. However, software involves not only development, but also other critical processes such as provisioning, operations, and use. In ways that will be described, it differs markedly from information as an economic good.

The lack of research from this general perspective is likely due to the complicated and sometimes arcane nature of software, the process of creating software, and the industrial processes and business relationships surrounding it. With this paper, we hope to rectify this situation by communicating to a broad audience, including especially the economics, business, and legal disciplines, the characteristics of software from an industrial and economic perspective. There are myriad opportunities to study software and software markets from a broader economic, industrial, and managerial perspective. Given the changing business models for creating and selling software and software-based services, it is an opportune time to do so.

1.1 Software—a unique good

Like information, software is an *immaterial* good—it has a *logical* rather than physical manifestation, as distinct from most goods in the industrial economy, which are material. However, both software and information require a material support infrastructure to be useful. Information is valued for how it *informs*, but there must be a material *medium* for storing, conveying, and accessing its logical significance, such as paper, disk, or display. Software is valued for what it *does*, but requires a computer *processor* to realize its intentions. Software most closely resembles a service in the industrial economy: a service is immaterial, but requires a provider (mechanical or human) to convey its intent.

Software differs markedly from other material and immaterial goods and services. On the supply side, its substantial economies of scale are much greater than material goods, with large creation costs but miniscule reproduction and distribution costs. In this regard, it is similar to information. On the demand side, unlike information (which is valued for its ability to influence or inform), software is similar to many material goods and to services in that its value is in the behaviors and actions it performs. In some circumstances, like computation, robotics, email, or word processing, it directly substitutes for services provided by human providers. In other cases, like the typewriter and telephone, it directly substitutes for material goods. Software is valued for its execution, rather than its insights. Additional understanding of software as a good will be developed later.

1.2 Software—a unique industry

In light of the uniqueness of software, the software industry has many characteristics that are individually familiar, but collected in unusual combinations. For example, like writing a novel, it is risky to invest in software creation, but unlike writing a novel it is essential to collaborate with the eventual users in defining its features. Like an organizational hierarchy, software applications are often essential to running a business, but unlike an organization, software is often designed by outside vendors with (unfortunately) limited ability to adapt to special or changing needs. Although software is valued for what it does, like many material goods, unlike material goods it has practically no unit manufacturing costs, and is totally dependent on an infrastructure of equipment providing its execution environment. To a considerably greater degree than most material goods, a single software application and its supporting infrastructure are decomposed into many internal units (later called modules), often supplied by different vendors and with distinct ownership. Even the term "ownership" has somewhat different connotations from material goods, because it is based on intellectual property laws rather than title and physical possession.

These examples suggest that the software industry—as well as interested participants like the end-user, service provider, and regulatory communities—confronts unique challenges, and indeed it does. In addressing these challenges, it is important to appreciate the many facets of software and how it is created and used in the real world, and that is the goal of this paper. The authors are technical specialists with a special interest in industrial, business, and economics issues surrounding software. We are especially interested in how software technology can be improved and the business and organizational processes surrounding more successful. To aid software professionals and managers, we hope to stimulate the consideration of superior (or at least improved) strategies for software investments. We believe it would be to the benefit of the software industry for economists and business strategists to study software's characteristics and surrounding strategies in greater breadth and depth. Thus, our primary goal is to aid the understanding of software as a good, as distinct from material and information goods, and to understand the processes surrounding it. We do not specifically address here the strategic challenges that flow from this understanding, but hope that this work will stimulate more and better investigation of this type.

1.3 Foundations of information technology

We all have an intuitive understanding of software based on our experience with personal computers. It is embodied by a "program" consisting of many instructions that "execute" on a computer to do something useful for us, the user. Behind the scenes, the picture is vastly more complicated than this, especially as software becomes an integral foundation of the operation of organizations of all types, and even society more generally.

Information technology (IT) is created for the primary purpose of acquiring, manipulating, and retrieving *information*, which can be defined as recognizable patterns (like text, pictures, audio, etc.) that affect or inform an individual or organization (group of people with a collective purpose). Information technology has three constituents: *Processing* modifies information, *storage* conveys it from one time to another, and *communications* conveys it from one place to another.

Often products valued for their behavior or actions have a material or hardware embodiment. *Hardware* refers to the portion of in information technology based directly on physical laws, like electronics, magnetics, or optics[1]. In principle, any information technology system can be constructed exclusively from hardware[2]. However, the central idea of computing is hardware *programmability*. The functionality of the computer is determined not only by the hardware (which is fixed at the time of manufacture), but can be modified after manufacture as the software is added and executed[3]. Since there is a fundamental exchangeability of hardware and software—each can in principle substitute for the other—it is useful to view software as *immaterial hardware*. The boundary between what is achieved in software and what in hardware is somewhat arbitrary and changes over time[4].

Fundamentally, information comes in different media, like a sound (as a pressure wave in the air), picture (a two-dimensional intensity field), or text (a sequence of alphabetical characters and punctuation marks). However, all information can be represented[5] by collections of *bits* (immaterial entities assuming two values: zero and one), such a collection is also known as *data*[6]. In information technology systems, all information and software are represented by data—this is known as a *digital* representation. This is advantageous because it allows different types of information, and even software, to be freely mixed as they are processed, stored, and communicated. IT thus focuses on the processing, storage, and communication of bits[7].

An operating IT system conveys streams of bits through time and space. Bits flow through space via a communications link from a sender to one or more receivers. Storage conveys bits through time: a sender stores bits at one time and a recipient retrieves these bits at some later time[8]. Processing modifies bits at specific points in space-time. When controlled by software, processing is performed by hardware specialized to interpreting the bits representing that software. A fundamental requirement is for material hardware to underpin all bit-level operations: the material structure (atoms, photons) brings the immaterial bits into existence and carries out the processing, storage and retrieval, and communication from one place to another.

1.4 Perspectives and issues

This paper views software from six individual perspectives, corresponding to the rows in Table 1: Users, software engineers, managers, lawyers, owners, and economists. We make no pretense of completeness in any of these perspectives, but focus on issues of greatest relevance and importance[9]. The main body of the paper is organized around these perspectives[10], in the order of the rows of Table 1.

We also focus on three basic issues, as reflected in the columns of Table 1. In *technology*, we address those technical characteristics of software and its execution environment that are especially relevant. One of the key distinctions here is between software *applications* (that provide functionality directly useful to end-users) and software *infrastructure* (that provides functionality common to many applications). *Processes* are the primary steps required to successfully supply, provision, and use software, and the precedence relationships among them. Finally, *value* considers the value-added of various functions and participants, and their interdependence. Specifically, there are two *value chains* in software, in which participants add value sequentially to one another. The *supplier* value chain applies to the execution phase, and starts with the software vendor and ends by providing valuable functionality to the user. The *requirements* value chain applies to the software implementation phase, and starts with business and application ideas, gathers and adds functional and performance objectives from users, and finally ends with a detailed set of requirements for implementation. Together, these two chains compose to form a *value cycle*[11]. Many innovations start with software developers, who are better able to appreciate the technical possibilities than users, but nevertheless require end-user input for their validation and refinement.

Altogether, there are many dependencies of technology, processes, and value. Some representative considerations at the intersection of perspectives and issues are listed in the table cells. Reference back to this table should be helpful in appreciating the relationship among the numerous issues addressed later. The following sections now consider the six perspectives (rows in Table 1) and how they relate.

Table 1. Examples of issues (columns) and perspectives (rows) applying to commercial software.

		Technology	Processes	Value
Participants	**Needs (users)**	Flexibility	Security, privacy	Functionality, impact
	Design (software engineers)	Representation, languages, execution, portability, layering	Architecture, composition vs. decomposition, standardization	Requirements, functionality, quality, performance
Facilitators	**Roles (managers)**	Infrastructure	Development, provisioning, operations	Uses
	Legal & policy (lawyers, regulators)	Intellectual property (patent, copyright, trade secret)	Licensing, business process patents, antitrust	Ownership, trademark (brand)
	Industrial organization (owners)	Components, portability	License vs. subscribe, outsourcing	Software and content supply, outsourced development, system integration, service provision
Observers	**Economics (economists)**	Costs	Business relationships, terms and conditions	Supply, demand, pricing

2 User perspective

The primary purpose of software is to serve the needs of its end users, whether they are individuals, groups of individuals, organizations (e.g. companies, universities, government), groups of organizations (e.g. commerce), or society at large (e.g. entertainment, politics).

To the user, the only *direct* impact of technology is the need to acquire, provision, and operate a complementary infrastructure to support the execution of the applications, which includes hardware and software for processing, storage, and communication. As discussed in Section 4, there are substantial organizational processes surrounding a software application, and a major challenge for both the end-user and vendors is coordinating the design and provisioning of the application with those processes, and/or molding those processes to the software.

Software informs a computer (rather than a person) by giving it instructions that determine its behavior. Whereas information embodies no behavior, the primary value of software is derived from the behavior it invokes; that is what it causes a computer to *do* on behalf of a user, and various aspects of how well it does those things. Although much depends on the specific application context, there are also important generic facets of value that are now discussed.

There are various costs associated with acquiring, provisioning, and operating software, including payments to software suppliers, acquiring supporting hardware, and salaries (see Section 4). Software with lower costs enhances the user's value proposition[12].

2.1 Productivity and impact

One way to value an application is the tangible impact that it has on an organization (or individual user) by making it more effective or successful. An application may improve the productivity of a user or organization, decrease the time to accomplish relevant tasks, enhance the collaboration of workers, better manage knowledge assets, or improve the quality of outcomes[13]. Applications can sometimes enable outcomes that otherwise would not be achievable, as in the case of movie special effects or design simulation.

2.2 Network effects

For many software products, the value depends not only on intrinsic factors, but also increases with the number of other adopters of the same or compatible solutions. This *network effect* or *network externality* [Sha99, Chu92, Kat85, Kat86] comes in two distinct forms [Mes99a]. In the stronger *direct* network effect, the application supports direct interaction among users, and the value increases with the number of users available to participate in that application. (In particular, the first adopter typically derives no value.) In the weaker *indirect* network effect, the value depends on secondary assets like available content or trained staff, technical assistance or complementary applications, and more adopters stimulate more investment in these secondary assets. An example of direct network effects is a remote conferencing application that simulates a face-to-face meeting, whereas the Web exhibits an indirect network effect based on the amount of content it attracts. An intermediate example would be a widely adopted word processing application, which offers substantial value to a solitary user, but also increases in value if many users can easily share documents.

2.3 Usage

Generally speaking, software that is used more offers more value. Usage has two factors: the number of users, and the amount of time spent by each user[14].

2.4 Quality and performance

Quality speaks to the perceptual experience of the user [Sla98]. The two most immediate aspects of quality are the observed number and severity of defects and the observed performance[15].

The most important performance parameters are the *volume* of work performed (e.g. the number of Web pages served up per unit time) and the interactive *delay* (e.g. the delay from clicking a hyperlink to the appearance of the requested page). Observed performance can be influenced by perceptual factors[16], but when the "observer" is actually another piece of software, then objective measures apply[17].

Perceived and real defects cannot be avoided completely. One reason is an unavoidable mismatch between what is built and what is needed. It is difficult enough to capture precisely the requirements of any individual user at any one point in time. Most software targets a large number of users (to increase revenue) and also needs to serve users over extended periods of time, during which their requirements change. Requirements of large numbers of users over extended periods of time can at best be approximated. Perceived defects are defined relative to specific requirements, which cannot be captured fully and accurately[18]. A second reason is the impracticality of detecting all design flaws in software[19].

These observations notwithstanding, there are important graduations of defects that determine their perceptual and quantifiable severity. For example, any defect that leads to significant loss of invested time and effort is more severe than a defect that, for example, temporarily disturbs the resolution of a display.

2.5 Usability

Another aspect of quality is *usability* [Nie00, UPA]. Usability is characterized by the user's perception of how easy or difficult it is to accomplish the task at hand. This is hard to quantify and varies dramatically from user to user, even for the same application. Education, background, skill level, preferred mode of interaction, experience in general or with the particular application, and other factors are influential. Enhancing usability for a broad audience thus requires an application to offer alternative means of accomplishing the same thing[20] or adaptation[21] [Nie93]. Like quality, usability is compromised by the need to accommodate a large number of users with different and changing needs.

2.6 Security and privacy

Security strives to exclude outside attacks that aim to unveil secrets or inflict damage to software and information [How97, Pfl97]. *Privacy* strives to exclude outside traceability or correlatability of activities of an individual or organization [W3CP]. Both security and privacy offer value by restricting undesirable external influences.

The details of security and privacy are defined by *policies*, which define what actions should and should not be possible. Policies are defined by the end-user or organization, and enforced by the software and hardware[22]. Often as these policies become stricter, usability is adversely impacted. It is therefore valuable to offer configurability, based on the needs of the individual or organization and on the sensitivity of information being protected.

A separate aspect of security and privacy is the establishment and honoring of *trust*. Whenever some transaction involves multiple parties, a mutual network of trust needs to be present or established, possibly with the aid of trusted third parties [Mess99a].

2.7 Flexibility and extensibility

Particularly in business applications, flexibility to meet changing requirements is valued[23]. Today, business changes at a rapid rate, including organizational changes (mergers and divestment) and changes to existing or new products and services.

End-user organizations often make large investments in adopting a particular application solution, especially in the reorganization of business processes around that application. Software suppliers that define and implement a well-defined roadmap for future extensions provide reassurance that future switches will be less necessary.

2.8 Composability

A single closed software solution offers less value than one that can be combined with other solutions to achieve greater functionality. This is called the *composability* of *complementary* software solutions. A simple example is the ability to share information and formatting among individual applications (like word processor and spreadsheet) in an office suite. A much more challenging example is the ability to compose distinct business applications to realize a new product or service.

3 Software engineering perspective

The primary function of software engineering is the development (which includes design, implementation, testing, maintenance, and upgrade) of working software [Pre00]. Whereas the user represents the demand side, software development represents the supply side. There are intermediaries in the supply chain, as detailed in Section 4. A comprehensive treatment of development would fill many books, so we focus on a few salient points.

3.1 Advancing technology

Processing, storage, and communications are all improving rapidly in terms of cost per unit of performance[24]. In each case, this improvement has been exponential with time, doubling in performance at equivalent cost roughly every 1.5 to 2 years and even faster for storage and fiber-optic communication. Continuing improvements are expected, with foreseeable improvements on the order of another factor of a million. Physical laws determine the *ultimate* limits, but the *rate* of improvement far short of those limits (as is the state of technology today) is determined by economic considerations. Technology suppliers make investments in technology advancement commensurate with current revenues, and determine the increments in technology advance based on expectations about increased market size, the time to realization of returns on those investments, and the expected risk. These factors all limit the rate of investment in research, development, and factories, largely determining the rate of technological advance[25]. A predictable rate of advancement also serves to coordinate the many complementary industry participants, such as microprocessor manufacturers and semiconductor equipment vendors[26].

These technology advances have a considerable impact on the software industry. Fundamentally, they free developers to concentrate on factors other than performance, such as features that enhance usability (e.g. graphical user interfaces and real-time video), reduced time to market, or added functionality[27].

3.2 Program execution

A software program embodies the actions required in the processing, storage, and communication of information content. It consists of the instructions authored by a programmer—and executed by a computer—that specify the detailed actions in response to each possible circumstance and input.

Software in isolation is useless; it must be executed, which requires a *processor*. A processor has a fixed and finite set of available instructions; a program comprises a specified sequence of these instructions. There are a number of different processors with distinctive instruction sets, including several that are widely used. There are a number of different execution models, which lead directly to different forms in which software can be distributed, as well as distinct business models.

3.2.1 Platform and environment

As a practical matter, consider a specific developed program, called our target. Rarely does this target execute in isolation, but rather relies on complementary software, and often, other software relies on it. A *platform* is the sum of all hardware and software that is assumed available and static from the perspective of our target. For example, a computer and associated operating system software (see Section 3.2.5) is a commonplace platform (other examples are described later). Sometimes there is other software, which is neither part of the platform, nor under control of the platform or the target. The aggregation of platform and this other software is the *environment* for the target. Other software may come to rely on our target being available and static, in which case our target is a part of that program's platform. Thus, the platform is defined relative to a particular target.

3.2.2 Portability

It is desirable that programming not be too closely tied to a particular processor instruction set. First, due to the primitive nature of individual instructions, programs directly tied to an instruction set are difficult to write, read, and understand. Second is the need for *portable execution*—the ability of the program to execute on different processors—as discussed in Section 3.5. For this reason, software is developed using an abstract execution model, divorced from the instruction set of a particular processor.

Portability of a given program means that full functionality is preserved when executing on different computers and operating systems. This requires that a new platform be created that appears uniform to our portable target program. Adding software to each operating system creates such a new and uniform platform. This new platform, often called a *virtual machine*, creates uniform ways to interact with operating system resources, input and output devices, and the network. It also creates a uniform representation for programs across different computers, enabling portable execution.

Particularly in the networked age, portability is an essential business requirement for many applications (see Section 3.5).

3.2.3 Compilation and interpretation

The program format manipulated directly by the software developer is called *source code*. It is written and read by people and also by various *tools* (programs performing useful functions aiding the software development process). One such tool is an automatic translator to another program format; the result of such an automatic transformation is called *object code*. The form of object code that is directly executed on the target processor is called *native code*. It is not necessary to directly translate from source to native code. Instead, a series of transformations can be used to achieve that goal—these transformations can even be *staged* to happen at different times and places [LL96].

Traditionally, a single transformation occurred either at the time of development (called compilation) or immediately prior to execution (called interpretation). *Compilation* allows the developer to transform the code once and deliver native code for one specific target processor. *Interpretation* allows transformation on the fly, at the time of execution, by the target processor. The primary distinction is that compiled object code can execute on a single target processor, whereas interpretation allows code to be executed without modification on distinct targets[28]. Portable execution can be achieved with multiple compilations, but requires a different software distribution for each target processor. Interpretation allows portable execution with a single software source distribution.

In a multi-stage translation, compilation and interpretation can be combined[29] as in the Java language[30]. This allows software to be distributed in a form that can execute on different targets, but retains some of the advantages of compilation, such as better performance optimization. For software that is executed multiple times on the same processor, interpretation incurs an unnecessary performance penalty that can be avoided by using *just-in-time compilation* (JIT), in which a compiler is invoked within the interpreter to compile some of the intermediate object code to native code. This technology can include online optimization, which actually improves the compilation by observing the local execution[31]. Current implementations of Java illustrate this[32] [Sun99, SOT00].

Interpretation and JIT compilation are important techniques to achieve execution portability. Interpretation can be avoided entirely without losing portability by always applying install-time or JIT compilation, as is the case with the common language runtime of the Microsoft .NET Framework. In a narrower definition of portability, interpretation and JIT compilation can also be used by platform vendors to allow software designed for another target be run on their platform, for example to allow Windows applications designed for a Pentium platform to execute on a Digital Alpha platform[33].

3.2.4 Trust in execution

An important issue is the implicit trust that a user places in an executing program [DFS98]. An untrustworthy program could damage stored data, violate privacy, or do other bad things. This places an additional consideration and burden on the choice of an intermediate object code format. Two different models are currently in use. First, using cryptographic technology, a user can verify that object code originated from a reputable software vendor, and further that it has not been modified[34]. Second, at execution time, it can be verified that code is not behaving maliciously and policies on what the code can and cannot do can be enforced[35].

3.2.5 Operating system

An application program never comprises the totality of the software executing on a particular computer. Rather, that program coexists with: an *operating system*[36], which provides an abstract execution environment serving to isolate the program from unnecessary details of the computer hardware (e.g. the particulars of how data is stored on disk), hides the reality that multiple programs are executing concurrently on the same computer (called *multitasking*), allocates various shared resources (e.g. memory and processor cycles) to those programs, and provides various useful services (e.g. network communications). The operating system is thus an essential part of any platform, along with the hardware.

3.3 Software development process

The primary process of interest to software engineers is *development*. Programs today have reached an order of size and complexity that warrants careful consideration of this process. Physical limitations (such as processing power and storage capacity) are no longer a significant limitation to what can be accomplished in software; rather, the most significant limitations relate to managing complexity, the development process, and limited financial resources.

3.3.1 Waterfall model

Recall that the requirements value chain taps the end-user's experience to ultimately define the requirements of a given software development. This is augmented by the *waterfall* model of development [Roy70], which defines a set of distinct phases, each adding value to the phase before. *Conceptualization* and *analysis* develop a vision, a detailed plan for development in sufficient detail to warrant investment, and a set of detailed requirements. *Architecture* and *design* use a "divide and conquer" approach to break the overall system into pieces that can be realized (somewhat) independently. These pieces can then be *implemented* and *tested* individually, followed by *integration* of the modules (making them work together) and *testing* and *evaluation* of the resulting functionality and performance.

Traditional development methods emphasized processes that start with end-user requirements and end with deliverable software, but this picture is now largely irrelevant. Instead, most new software results from a modification and update of existing software [Vac93]. Where the produced software is monolithic, the asset that enables production of new software is the established source base (the repertoire of source code available to *and mastered by* an organization). Software components (see Section 6.5.2) are an alternative complementary to maintaining source bases: instead of viewing source code as a collection of textual artifacts, it is viewed as a collection of units that separately yield components. Instead of arbitrarily modifying and evolving an ever-growing source base, components are individually evolved and then composed into a multitude of software products.

While the waterfall model is useful for identifying the distinct activities in development, it is highly oversimplified in practice because it does not recognize the existing code base[37], it does not recognize that these phases are actually strongly overlapping, and because requirements are rarely static throughout development.

3.3.2 Development tools

An additional source of value, because they greatly reduce the development time and cost, are development *tools*[38]. These tools automate tasks that would otherwise be time consuming, and do a number of other functions, such as keeping track of and merging changes. Sophisticated toolkits are necessary for the management and long-term success of large projects involving hundreds or thousands of software engineers.

3.3.3 Architecture

The notion of building software based on available assets can be moved to a more principled approach. Instead of relying on developers to 'discover' that some available code or component may be reused in new situations, software systems are designed such that they are related by construction. The level of design that emphasizes such relationships is *software architecture* [BCK98, Bos00]. Like tools, architecture plays an important role in containing the complexity of the system, in this case by allowing the overall system to be composed of pieces developed largely independently.

The primary role of architecture is to address system-wide properties by providing an overall design framework for a family of software systems. Concrete designs then fit in by following the architecture's guidelines and complementing it with concrete local design decisions. If done properly, architecture decomposes systems into well-identified pieces called *modules*, describes their mutual dependencies and interactions, and specifies the parameters that determine the architecture's degrees of configurability. As illustrated in Figure 1, architecture has three facets: the *decomposition* of the system into modules, the *functionality* of each module, and the *interaction* among modules. *Global system properties* (a.k.a. *system qualities*), such as performance, maintainability, extensibility, and usability, emerge from the concrete composition of modules[39] [CSA98].

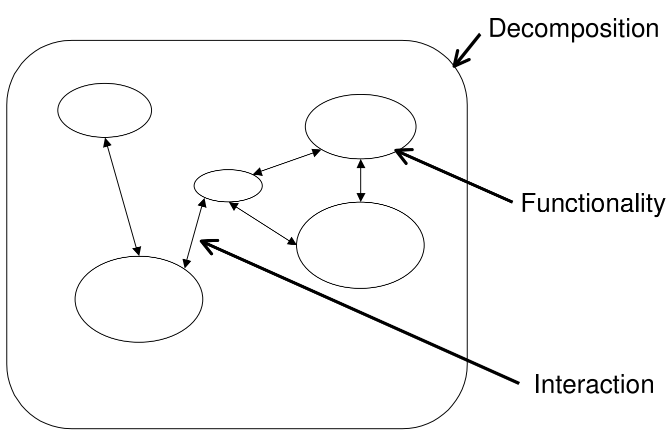

Figure 1. An illustration of a simple software architecture.

"Modular" is a term describing architectures that have desirable properties from the perspectives of supporting a good development methodology and containing complexity [Par72, Bak79, Jun99]. One key property is strong *cohesion* (strong internal dependencies within modules) and weak *coupling* (weak dependencies across module boundaries). Other desirable properties of modular architectures have become accepted over time[40].

As illustrated in Figure 2, modular architectures are usually constructed hierarchically, with modules themselves composed of finer-grain modules. This enables the same system to be viewed at different granularities, addressing the tension between a coarse-grain view (relatively few modules to understand) and a fine-grain view (small modules that are easy to implement). Of course, the cohesion of modules is inevitably stronger at the bottom of the hierarchy than at the top[41].

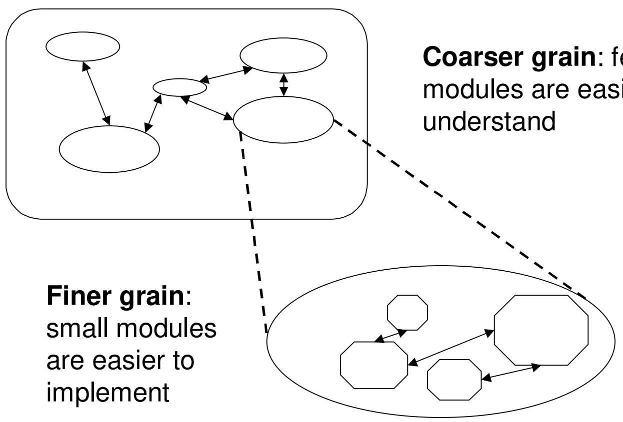

Figure 2. An illustration of hierarchical decomposition.

Software architecture has interesting parallels in the design of human organizations [Lan00, Lan92, Bal97, San96]. The principles of modularity can be applied there as well.

3.3.4 Interfaces and APIs

The interaction among modules focuses on *interfaces*. The module interface tells, roughly speaking, how other modules are to 'use' this module. More precisely, an interface specifies a collection of atomic[42] *actions* (with associated data parameters and data returns) and *protocols* (compositions of actions required to accomplish specific ends). Multiple protocols may share a given action.

The second purpose of the interface is to inform the module developer as to what must be implemented. Each action is implemented as an operation on internal data, and often requires invoking actions on other modules. Importantly, an interface is designed to hide irrelevant internal implementation details so that the latter can be freely changed without other modules becoming dependent on them[43]. The *encapsulation* of implementation details precludes bypassing the interface and creating unnecessary (even inadvertent) dependencies[44].

An interface meant to accept a broad and open class of extensions—modules that are added later, following deployment—is called an *application-programming interface (API)*[45].

3.3.5 Achieving composability

There are two distinct approaches to modular software development. In *decomposition*, modules are defined in response to the required system functionality, and in *composition*, that functionality is achieved by composing pre-existing modules. Composition is the focus of component software, discussed in Section 6.5.

Architecture and development focus on defining and implementing modules that can later be composed (see Section 2.8). The additional functionality that arises from composition, called *emergence*[46], is a source of value in the development stage of the supply chain. While critically important, composability is actually difficult to achieve, although it is considerably easier for top-down decomposition than for bottom-up composition. It requires two properties: interoperability and complementarity.

For two modules to communicate in a meaningful way, three requirements must be met. First, some communication infrastructure must enable the physical transfer of bits[47]. Second, the two modules need to agree on a protocol that can be used to request communication, signal completion, and so on. Finally, the actual messages communicated must be encoded in a mutually understood way. Modules meeting these three requirements are said to be *interoperable*.

Mere interoperability says nothing about the meaningfulness of communication. To enable useful communication, the modules need to *complement* each other in terms of what functions and capabilities they provide and how they provide them. (An example of non-complementarity is the failure of a facsimile machine and a telephone answering machine to cooperate to do anything useful, even though they can interoperate by communicating over the telephone network[48].) Modules that are interoperable *and* complementary (with respect to some specific opportunity) are said to be *composable* (with respect to that opportunity). Composable modules offer additional value since the composed whole offers more functionality and capability than its pieces. The Web browser and server offer a rich example of interoperability[49], complementarity[50], and composability[51]. Usability (see Section 2.5) can be considered a form of composability of the user with the software application.

3.4 Software as a plan and factory

The question arises as to the character of software as a good. Is it similar to information goods in the “new economy”, or material goods in the “industrial economy”? We have pointed out that the demand for software differs from information in that it is valued for what it does, rather than how it informs[52]. Many goods in the material world are valued for what they do (e.g. the automobile, which takes us places), so, the question arises: Is software perhaps closer in its characteristics to many material products (traditional engineering artifacts) than to information? From the perspective of the user, on the demand side, it is[53]. However, in terms of its development, the supply-side, one property sets it far apart.

If a software program were analogous to a material product or machine, we could view it as a predefined set of modules (analogous to the parts of a material machine) interacting to achieve a higher purpose (like the interworking of parts in a machine). If this were accurate, it should be possible, to a greater extent than is realized today, to construct software from standard, reusable parts—the “industrial revolution of software”.

This view is incorrect. In fact, the set of interacting modules in an executing program is *not* pre-defined. During execution, a large set of modules are created dynamically and opportunistically based on the particular needs that can be identified only at that time. An example would be a word processor, which often creates literally millions of modules at execution time tied to the specific content of the document being processed[54]. The programmers provide the set of available modules, and also specify a detailed plan by which modules are created dynamically at execution time[55] and interact to achieve higher purposes.

Programming is analogous to creating a plan for a very flexible factory in the industrial economy. At execution, programs are universal factories that, by following specific plans, manufacture an extremely wide variety of immaterial artifacts on demand and then compose them to achieve higher purposes. Therefore, a program—the product of development—is not comparable to a hardware product, but rather more like a factory for hardware components, and one that is highly flexible at that. The supply of raw materials of such a factory corresponds to the reusable resources of information technology: instruction cycles, storage capacity, and communication bandwidth.

In short, software products are most closely analogous to a plan for a very flexible factory on the supply side, and to a material product (created by that factory) on the demand side. The plan is a form of information—and one that shares many characteristics of information like high creation costs and low reproduction costs—but it informs the factory (executing program) rather than the consumer.

Other engineering disciplines are similarly struggling when aiming at methods to systematically create new factories, especially flexible ones [Upt92]. The common belief that software engineering has yet to catch up with more mature engineering disciplines is thus exaggerated.

3.5 Impact of the network

The spectacular success of the Internet has had a dramatic impact on software. It enables *distributed* applications composed of modules executing on different computers interacting over the network. Distributed applications that can execute across heterogeneous platforms serve a larger universe of users, and due to network effects offer greater value[56]. While portability was useful before, because it increased the available market size for software vendors, it becomes much more compelling in the networked world. The network also makes interoperability more challenging, because interacting modules are more likely to come from different vendors, or to be executing in heterogeneous administrative environments (like across organizational boundaries) with less opportunity for coordinated decision-making.

The network offers another major opportunity: Software programs can be transported over the network just like information, since they can be represented by data. This offers an attractive distribution channel, with low cost and delay[57].

Traditionally software is semi-permanently installed on each computer, available to be executed as needed. The idea with *mobile code* is to opportunistically transport a program to a computer and execute it there, ideally transparently to the user[58]. By eliminating pre-installation of software, mobile code can help overcome network effects by transporting and executing applications all in a single step, avoiding the need for pre-installation and the difficulty in achieving interoperability between different releases of a given program. Mobile code can also move execution to the most advantageous place; e.g. near the user (enhancing responsiveness) or where there are available resources[59].

While mobile code enables the opportunistic distribution of code, in some circumstances it is necessary for a program to actually move between processors during the course of its execution. This is called a *mobile agent*, and requires that the program carry its data[60] as well as code with it. Mobile agents have applications in information access and negotiation, but also pose challenging security and privacy challenges.

The multi-stage translation described in Section 3.2 is important for networked software distribution and mobile code. As discussed later, it is rarely appropriate for business reasons to distribute source code, and native object code is problematic on the network because of the heterogeneous platforms (although standard for “shrink-wrapped” software products). Hence, an intermediate form of object code becomes the appropriate target for software distribution, relying on compatible interpreters on each platform[61].

3.6 Standardization

An *open industry standard* is a commonly agreed, well-documented, and freely available set of specifications, accompanied by no intellectual property restrictions (or possibly restrictions that are not burdensome and are uniform for all). (The opposite case would be proprietary specifications not made available to other vendors.) Especially as a means of achieving interoperability over the network, where software from different vendors must be composed, complicated by heterogeneous platforms, standards become an essential enabler[62]. Thus, standards processes become an essential part of the collective development activities in the networked software industry [Dav90]. In addition, users and managers encourage open standards because they allow mixing and matching of different products, encouraging both competition and specialization of industry, with advantages in availability, cost and quality.

As applied to interfaces, the purpose is to allow modules implemented by different software vendors to interoperate. The first step in any standards effort is to define the decomposition of the overall system into typical modules: this is called a *reference model*[63]. A reference model is a partial software architecture, covering only aspects relevant to the standard[64]. The standards process can then specify the functionality and interfaces of the modules, insuring composability[65]. Another common target of standards is the data representation for common types of information, such as documents (e.g. HTML used in the Web and MPEG for video). De facto standards, which arise through market forces rather than any formal process, are interfaces or data representations that are widely used[66].

Standards also address a serious problem in software engineering. In principle, a new interface could be designed whenever any two modules need to compose[67]. However, the number of different interfaces has to be limited to reduce development and maintenance costs. Besides this combinatorial problem, there is the open world problem. The open world assumption in systems allows new modules to be added that weren't known or in existence when the base system was created. It is not only impractical but also clearly impossible to have a complete set of special-case or proprietary interfaces that connect a full range of modules that may arise over time.

Interfaces, the functionality related to these interfaces, the preferred decomposition of systems into extensions, and the representations used for data crossing the interfaces all need to be standardized to enable interoperability. For needs that are well understood and can be anticipated by standardization bodies (such as industrial consortia or governmental standardization institutions) standards can be forged in advance of needs and then implemented by multiple vendors. This approach has had a tendency to fail outright or to be too slow and cumbersome when the attempted standardization was simultaneously exploring new territory. This has led to new standardizations processes well integrated with a research endeavor, such as the Internet Engineering Task Force (IETF)[68].

An approach to standardization called *layering* allows standards to be built incrementally and enhanced over time, rather than defined all at once[69] (the IETF follows this approach). The first layer, called wiring or plumbing standards, is concerned with simple connection-level standards. As with other types of wiring or plumbing, it is entirely feasible to establish connections at this level that are meaningless (or even harmful) during composition. Standardization can then be extended one layer at a time, establishing ever-richer rules of interoperation and composability.

4 Managerial perspective

Software presents severe management challenges, some of them relating directly to the software, and some relating to the organizational context of the software application.

The supplier value chain from software vendor to user has four primary stages, as listed in the rows of Table 2: development, provisioning, operation, and use[70]. Each of these roles presents management challenges, and each adds value and thus presents business opportunity as discussed in Section 6. The *development* stage involves not only initial design and implementation (as described in Section 3.3), but also the ongoing maintenance and upgrade of the software. In the *provisioning* stage, the facilities (network, servers, PC's) are purchased and deployed, depending in large part on performance requirements, and the software is installed, integrated, and tested. At the *operations* stage, an application and its supporting infrastructure is kept running reliably and securely. At the *use* stage, the application functionality provides direct value to users and end-user organizations (as discussed in Section 2).

Table 2. Stages of the supplier value chain (rows) vs. generic tasks (columns).

	Planning	**Deployment**	**Facilitation**	**Maintenance**	**Evolution**
Development	Functional and performance requirements	Build systems	Software tools support	Defects repair, performance tuning	Tracking requirements, upgrade
Provisioning	Organizational design, performance requirements	Installation, integration, configuration, and testing	Procurement, finance		Installation, integration, configuration, and testing
Operation			Systems administration	Patching	
Use	Organization	Organizational adjustments, training	Help and trouble desk		Organization and training

4.1 Four value chains

There are two distinct types of software, and hence in reality two supplier value chains. Application software provides specific functionality meeting the needs of end users, and infrastructure provides generic capabilities subsumed by many applications. The infrastructure includes both hardware (computers, peripherals, and communications links and switches) and software[71]. Examples of the latter are the operating system and a general category called middleware (discussed later). Infrastructure does not provide *direct* value to the user, but is essential to the operation of an application.

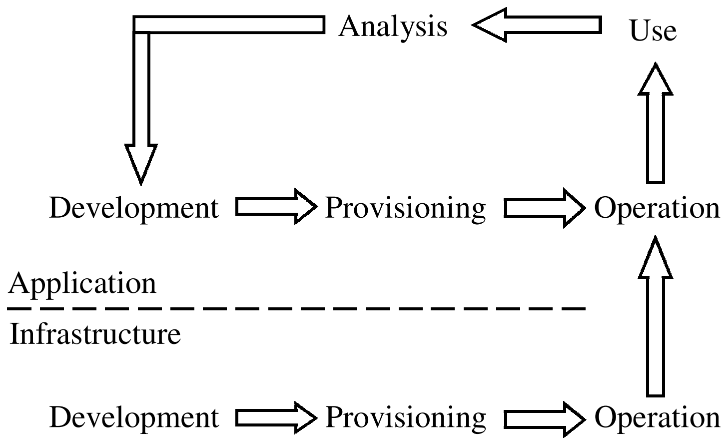

Figure 3. Three value chains in the software industry.

Considering the separation of application from infrastructure, there are two supplier value chains and one of the two requirements value chains, as illustrated in Figure 3. The development, provisioning, and operation of infrastructure adds value indirectly by enabling applications, as well as making them easier to develop, provision, and operate. The development, provisioning, and operation of the application provide direct value to the user. The requirements chain from the user adds value to the application development by defining appropriate requirements that better meet user needs. User requirements have no direct relevance to the infrastructure, which is designed to serve many applications. However, the collective needs of many applications form a second requirements value chain (not shown).[72].

4.2 The stages of the supply value chain

The columns of Table 2 show some generic issues that require management attention, and the cells in the table list specific roles for each stage in the value chain that contribute to resolving these issues. In many cases these roles correspond to specific job functions, with specialized training and skills. The four stages in the supply value chain are now discussed in greater detail.

4.2.1 Development

The responsibilities of a software developer continue throughout the product's lifecycle, that is, as long as the software is supported. Other players at all stages require *support*, answering inquiries and assisting with the location of problems or its successful use. An ongoing *maintenance* role (in the form of *service packs* or *patches*) is the fixing of reported flaws. Also, all software requires *upgrades*, (in the form of periodic *releases*) with often extensive re-programming, to fix flaws, and meet changing requirements or add new features.

4.2.2 Provisioning

Provisioning includes the selection, negotiation, and purchase (or other arrangements, such as licensing, leasing, or subscription) of all the facilities (equipment and communication links) and software required to execute an application and its supporting infrastructure. Often this involves a design element: the sizing of the facilities to meet the performance needs of the users[73]. Provisioning also includes the actual installation and testing of the resulting equipment and software, with acceptance criteria based on functionality and performance criteria. Often the communication and information processing (processing and storage) portions of the facilities have separate provisioning processes. Either or both of these elements may be outsourced to a firm providing this as a service called *systems integration.*

4.2.3 Operations

The daily operation of most software systems requires some degree of attention. For example, with organizational and personnel changes, authorization levels need to be adjusted. Security is another issue that requires vigilant attention, and patches fixing security holes or flaws must be installed. Together, these functions are called *system administration*. In addition, the facilities need to be adjusted and reconfigured to meet changing organizational needs and changing distributions of workload. This is called *system management*, and in a sense is an extension of the provisioning phase. Together administration and management are critical factors that determine to a significant degree an application's effectiveness and efficiency.

4.2.4 Use

End-user organizations perform a number of important support functions for individuals using applications[74]. During the planning, important issues revolve around business processes and organizations that use the application. In the planning stage, an important issue is whether to develop a custom application (thus molding it more closely to business processes) or a common off-the-shelf (COTS) application (requiring changes to match the assumptions incorporated in the design). The vendors of COTS applications try to make them as configurable and parameterizable as possible[75], providing greater flexibility but also necessitating considerable effort in the provisioning phase[76].

Preparatory to the operations phase, the training of workers to properly use the application as well as execute other elements of the business process is critical. During operations, an organization must also provide help to its users, and provide a point of contact for problems (commonly called a 'helpdesk').

4.3 Total cost of ownership

An important consideration to managers is the *total cost of ownership* (TCO) of an application. This includes the cost of provisioning, operations, and user support. In addition, it may include development and maintenance costs for an application developed and maintained internally. In cases where users must perform administrative functions (like administrating their own desktop computers) or provide training of or help to other users, the imputed costs of these responsibilities should be included in the TCO.

As the TCO has become an appreciable part of a typical organization's budget, reducing the TCO has become a significant issue to managers and suppliers. The quest to lower the TCO has resulted in pressure on vendors to provide streamlined administration and management of applications and infrastructure, as well as improved usability with simplified training and help requirements. The observation that considerable costs result from the administration of desktop computers (including by users as well as systems administrators) has resulted in a trend toward greater centralization. In particular, moving the points of administration and management from desktop computers (called *clients*) to centralized computers (called *servers*) where, at minimum, fewer computers need be administered and managed, can reduce costs. In a sense, this harks back to the days of centralized mainframes[77], albeit with considerable differences[78]. An extreme case is *thin clients*, where the desktop computer executes no application specific code except that which can be dynamically loaded as mobile code. An alternative is *rich clients* (rich in local customizability and functionality) supported by improved centralized administration and management mechanisms. Most organizations deploy a mixture of thin and rich clients, reflecting the varying job profiles supported by the client systems.

5 Legal perspective

The legal system plays an important role in establishing and enforcing property rights for software. Increasingly, government regulation is contemplated to address issues surrounding software, such as privacy, control over access (particularly for children), and controls over the use of encryption[79].

5.1 Copyright

Like information, exact replicas of software are easily created, and these replicas are non-rival in use. This easy replication makes unauthorized copying and distribution trivial. Security schemes to discourage this also inhibit usability and thus encounter strong customer resistance. Only social constructs such as established ethics, legal restrictions and active law enforcement can prevent *piracy*—large-scale unauthorized manufacture and sale of software—and, it is argued, encourage substantial investments in the creation of software[80].

The copyright protects an original creation of software by granting the creator exclusive control (including the right to sell or license) of the work and precluding others from appropriating, replicating, and selling the software without permission. It does not prevent others from independently developing a similar work based on the same ideas or intended purposes. The original developer can also maintain control over derivative works, such as new releases.

Software is normally licensed to whoever provisions and operates it. The license can contain arbitrary terms and conditions on use, payment, and dissemination—including timing and size of payments. Because the replication (analogous to manufacturing) costs are low, some unusual licensing terms become economically viable. *Freeware* involves no payment and allows the user to replicate and distribute the software freely. *Shareware* is provided for free, but a voluntary payment is requested if the user puts the software to productive use[81]. *Copyleft* encourages derivative works, but requires that they themselves be freeware or copyleft.

Copyrights can protect the property rights of either source or object code, but it is almost always object code that is distributed and licensed. Object code places an additional obstacle to reverse engineering to uncover trade secrets (proprietary ideas or methods)[82]. Most important, object code is much more difficult to modify—which is important because customer modifications would effectively invalidate warranties and preclude customer support[83]. Distributing object code also contributes to the encapsulation of implementation details, avoiding unnecessary dependencies in any composition with other software.

Open source is a form of freeware in which source rather than object code is released[84]. Associated with open source is usually an informal (but coordinated) group of dedicated volunteers who maintain and upgrade it.

5.2 Patents

A *patent* grants limited-term exclusive rights to make, use, or sell products incorporating an invention (roughly, a novel, non-obvious, and practically useful idea). Unlike the copyright, the patent owner can preclude others from using an invention, even if they discover it independently. Patents have only recently been applicable to software[85]. Patents have also recently been permitted on business processes, which often underlie software applications[86].

An invention is implicitly divulged as it is used—or at least the possibilities are revealed—and it is non-rival in use, making it easily appropriated and a poor investment without appropriate property rights. Patents encourage investment in research and development by promising a period of exclusivity, but also publicize inventions (rather than keep them a trade secret), allowing others to improve upon them even if they are not used.

6 Ownership perspective

The organization of the software industry depends strongly on technology, processes, and value. This organization of the marketplace into cooperating and competing firms is essentially an issue of ownership coupled with appropriate business relationships. Ownership and the right to derive monetary value provide the incentives to produce software[87]; an industrial and societal organization that honors these rights is thus key to effective production. Software architecture is strongly influenced by industrial organization, and vice versa, as the only practical boundaries of individual firms correspond to well-specified module interfaces. As for process and value, the stages of the value chain as defined in Section 4 form natural business functions with separate ownership, sometimes broken down further into specific job functions. Of course, these functions can also be bundled together as combined businesses.

6.1 Industrial organization

Since companies tend to form around individual units of value that enhance internal synergies and exploit common expertise, industrial organization [Rob95, Lan92] can be thought of as a partitioning of the value chain into distinct companies. In these terms, there are natural businesses formed by partitioning of the value chain of Figure 3 as illustrated in Figure 4.

The *application software supplier* typically bundles the analysis and development functions, working closely with the end-user organization to define requirements. Similarly, the *infrastructure software supplier* must be cognizant of the requirements imposed by a wide range of applications.

The *system integrator* specializes in provisioning. This role takes responsibility for acquiring software from both application and infrastructure suppliers (usually more than one), does whatever is necessary to make it work together, and installs and tests the software[88]. Some programming is typically involved as well[89]. Another role in provisioning is the *consultant*, who helps the end-user organization rework the organization and business processes around the software, and often helps configure the software to the needs of the particular end-user.

Operation is the specialty of a service provider. An *application service provider* (ASP) licenses and operates the application, while an *infrastructure service provider* (ISP[90]) purchases or licenses and operates the hardware and software infrastructure (computers, storage, network, operating system, etc.).

Of course, different partitions of the value chain are possible. A common bundling is an end-user organization that performs one or both[91] the service provider functions internally in an information systems (IS) department. A large organization may even develop at least some of its own applications, and serve as its own systems integrator. Other common bundles are an ISP that handles its own systems integration, and a service provider that operates both application and infrastructure[92]. The ISP function is often fragmented into two or more companies (e.g. separate backbone network and processing and storage service providers) or between the end-user organization and a service provider (e.g. internal application hosting and networking but a separate backbone network provider). A software developer may also become an ASP, effectively changing its business relationship with the end-user from one of licensing object code to providing application services by subscription.

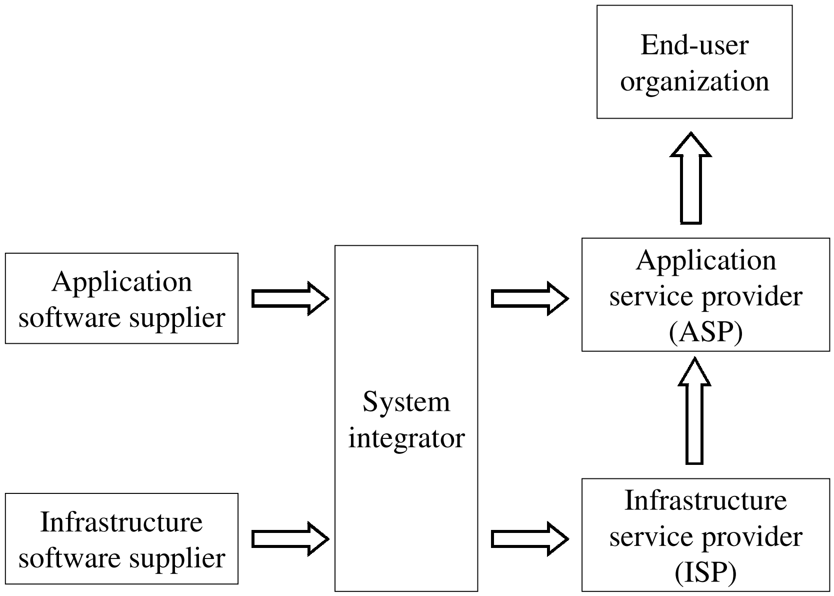

Figure 4. Natural businesses partitioning of the value chain.

Not shown in Figure 4 is the information content supplier. One focus of applications is the manipulation and presentation of information to the user, and this information may come from an independent source (e.g. a stock analyst that discloses company prospects to the users of a stock brokerage application).

Increasingly, the industry is evolving toward more complex industrial organizations in which applications are composed from two or more unbundled modules, which interoperate using the network. An example is *Web services*, where one Web-based application serving a user directly incorporates capabilities from another, using the Web infrastructure. In this case, the 'customer' of the latter is another piece of software, rather than the end-user. Alternatively, the first can be considered in part an intermediary on behalf of the user and the second, often adding value (by measures such as customization, aggregation, filtering, and consolidation).

6.2 Business relationships

Whatever the industrial organization, there are many possible business relationships among the participating firms. A selection of important issues is now discussed.

6.2.1 Types of customers

Today, virtually every organization, and a large percentage of individual citizens, are customers of software vendors or are themselves software developers. Users come in four distinct categories: Individuals license applications for their own purposes, such as personal productivity, collaboration, information access, and entertainment. Organizations license, purchase, or develop internally applications that support their internal business and external business relationships. Original equipment manufacturers (OEMs) embed software within equipment that they manufacture and sell[93]. Finally, the 'customer' of a piece of software can be another software application, as in Web services.

6.2.2 Software distribution

There are several ways to get software to a customer. As explained in Section 5.1, software is normally distributed as object code. That object code is represented in a binary alphabet, and can be distributed on magnetic or optical media or over a network.

The network as a distribution channel for software is increasingly important. It is inexpensive, especially when used to bypass the normal distribution chain, and timely. These properties make it valuable for the frequent distribution of maintenance and new releases, giving greater freedom to change applications with less fear of incompatibilities[94]. However, intermediaries remain useful in the distribution chain, particularly where there are large numbers of alternative suppliers to consider, or where the integration and bundling of different products is needed.

What has to happen before a customer can execute software? There are at least four possibilities: First, the software may have been embedded (pre-installed) in a piece of equipment before that equipment is distributed. Such equipment, when sold to an end-user, is called an *appliance*. Second, the customer may have to install the software herself, which requires conspicuous action (this is *user self-provisioning*). Third, the software may download over the network and simply execute without an installation step. This is called *mobile code*. Fourth, the customer may simply use software executing remotely, which is operated by an ASP. From the customer perspective, these options are similar, with the exception of the second[95].

For user-installed or mobile software, a traditional business model of productization and marketing is appropriate. For embedded and ASP/ISP-operated software, the OEM or service provider acquires and provisions the software[96]. The key difference to the supplier is one of scale and sophistication: a small number of sophisticated OEMs or service providers as opposed to a large number of end-users. The decision process is also different: with an OEM or service provider or mobile code, a third party makes decisions on behalf of the end-user to install a new version or move to a competitive offering.

Mixtures of these distribution models are common. Appliances may fetch automatically installed upgrades over the network. An ASP may use mobile code to move a portion of the execution closer to the end user, thereby improving interactivity[97]. Similarly an ASP may require a complementary user-installed piece of software.

6.2.3 Software pricing

There are many alternatives and issues in designing pricing models (see Section 7.3). However, there are also fairly standard practices observed in the industry today, and they differ across distribution models. User-installed software is usually sold for a fixed price like traditional products. This places a premium on selling new releases to maintain a steady revenue stream, especially after a high penetration is reached[98]. This also makes its own installed base the main source of competition for the supplier.

OEM or service-provider models place an intermediary in the value chain, requiring two pricing strategies (software supplier to intermediary, and intermediary to end-user). From the perspective of the software supplier, a common approach is to approximate the user-installed pricing model by basing the price on the rate of end-user adoption (such as a fixed price per appliance sold, or proportional to the number of customers an ASP attracts)[99]. Other possibilities are opened up in the ASP case by the relative ease of metering other metrics of usage, such as the total number of transactions completed.

ASP pricing to an end user may use models commonly found in traditional service industries. Three dominant models are subscription, pay-per-use, and cross-subsidy. A *subscription* offers the service for capacity-limited or unlimited use over a contracted time period. *Pay-per-use* requires metering and billing on a per-use basis[100]. *Cross-subsidy* recovers the cost of providing a service (possibly incurring a loss or a profit margin) by attaching to a technically unrelated model, such as advertising or bundling with another paid service.

Another example of a distribution and payment model is the s*uperdistribution* [Cox96], which encourages anyone to further distribute software modules they find useful, with a mechanism and infrastructure for payments (typically based on metering and usage) to flow to the owner[101].

6.2.4 Acquiring applications

An end-user organization that is acquiring software for internal purposes has several options. Generally, these can be summarized as make, buy, license, and subscribe. Each has its advantages and disadvantages.

In the *make* option, all four stages (development through use) are kept in-house. This has the greatest potential for competitive differentiation, but offers no potential to amortize costs over multiple end-users, and invokes considerable delay and risk[102]. The *buy* option is to outsource development to a firm specializing in such developments. In this case, the source code may become the property of the end-user organization, and ongoing maintenance and upgrade may be the responsibility of the developer or user. Both options offer the opportunity to mold business processes, organizations, and the software as a unit, gaining efficiency and competitive advantage.

In the *license* option, an end-user licenses a software product from a software supplier. Finally, in the *subscription* option, the application services are purchased directly from an ASP. These are generally low-cost options, but offer little opportunity to differentiate from competitors. In both cases, the software largely defines business processes and organizations, and a consultant is frequently engaged to assist in making needed changes to processes and organizations[103].

6.2.5 Acquiring infrastructure

Sometimes, application software that becomes both ubiquitous and frequently composed into other applications effectively moves into the infrastructure category. For example, the Web was originally conceived as an information access application for scholarly communities [W3C95], but has evolved into an infrastructure supporting e-commerce and other applications. Another example is productivity application suites that today are commonly used as infrastructure for custom applications that require core functionality such as word processing or spreadsheets. In other cases, infrastructure is explicitly developed, either for a particular class of applications or for all applications.

An important distinction can be made between infrastructure that is a platform and infrastructure that is specialized to serve some applications. Different supplier firms tend to specialize in different types of infrastructure, and different business models apply. Some infrastructure is provided solely to simplify and reduce the cost of application development, and such infrastructure may be bundled with an application and licensed as a unit. In this case, the effect is to subsidize the infrastructure development cost with income generated by applications. This still relies on platform support to avoid repeated redundant or even conflicting installation of infrastructure software.

Since a platform must pre-exist any deployed application, and supports many applications, and because it is usually a massive undertaking, it is sold separately. This way, it can be licensed and installed only once. More importantly, separate applications often must be composed for higher-level functionality, like information sharing. One important value added (and economic underpinning) of a platform is interoperability among applications, which would not be possible if each application was bundled with its own infrastructure.

Infrastructure is rarely made or bought, but can be licensed or subscribed. Wide-area networking and communication services in particular are subscribed, because it is largely impractical for end-user organizations to provision their own communication lines and because a public network offers a richness of connectivity that would be impractical to match with dedicated facilities.

Consistent with the rising popularity of application subscription, there are indications that infrastructure offerings by subscription will grow in both richness and in popularity. An early example is caching, which improves the performance of information distribution by locating temporary storage of information nearer to end-users accessing that information[104].

6.3 Vertical heterogeneity

Like other industries, software has both vertical and horizontal heterogeneity that impacts the industry structure and the nature of business relationships. The vertical heterogeneity creates dependencies and complementarities among different types of firms. For example, application software has a vertical relationship to infrastructure software—it is dependent upon it[105]. Such dependencies also arise *within* the infrastructure—in software, this is called *layering* and is related to the layering of standards discussed in Section 3.6.

Layering is a specific architecture in which modules share a vertical relationship: Each layer is dependent on the layers below[106]. The layers are thereby complementary, and all layers are necessary to support an application. An example of infrastructure layering is shown in Figure 5. Lower layers are specific to the technology component (processing, storage, and connectivity), and provide common representations (how information elements are represented by bits[107]) and services (standard functions performed by the infrastructure[108]). Above this are integrative layers that bring together the services of the constituent technologies in useful ways[109]. Finally, on top lie the application components (discussed later) and applications themselves.

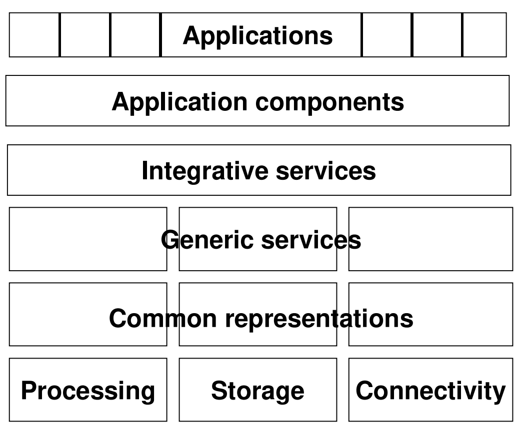

Figure 5. Internal software architecture.

It is desirable for a diverse set of applications to co-exist with a diverse set of core technologies[110], so that free market entry and innovation can be maintained in both applications and core technology with minimal dependence of the two. The key idea embodied by the middle infrastructure layers is, as illustrated in Figure 6, to define a set of common and universal representations and services. The applications speak to these common elements, which can be re-implemented for each new technology[111].

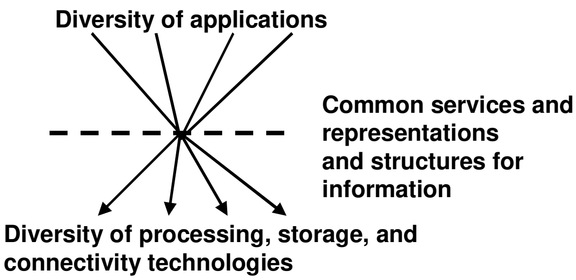

Figure 6. A separation of technological progress from applications.

The modern goal of each and every layer is to provide representations and services that are sufficiently general and configurable that they form a platform suitable for a wide variety of applications. This is a relatively new goal, and has not been totally achieved as yet. Formerly, the industry formed vertical stovepipes[112] for narrower classes of applications (e.g. separate infrastructures for voice, data, and video distribution) [Bro98]. This old model proved unsuitable for stimulating a diversity of applications, because of the necessary investment in an entirely new infrastructure for each new class of application[113].

The transition from stovepipes to layers has profound implications for industry structure. While stovepipes like the mainframe, UNIX server, PC, and telephone networks resulted in largely independent marketplaces, suddenly companies must adapt to a world in which they support all applications. Specialization tends to occur at the individual layers, rather than being associated with narrower classes of applications. This moves away from vertical integration and towards horizontal integration, and no single company can provide a complete integrated solution to the customer. Competition forms at the individual layers, and customers (or a system integrator or intermediary) must integrate products together for a complete solution.

6.4 Horizontal heterogeneity

The ideal of a set of homogeneous infrastructure layers is not quite reality because of attempts of suppliers to differentiate themselves and other practical difficulties. The more severe difficulty results from networking: Distributed applications are partitioned and deployed to multiple hosts across the network. This introduces some significant operational challenges, and affects business relationships.

6.4.1 Multiple platforms

As a result of historical trends and industry competition, there is horizontal heterogeneity in the platforms. For example, a number of operating system and processor platforms (e.g. the IBM mainframe, SUN/Solaris, Macintosh, and PC/Windows) coexist. Post Internet, there is need for applications to execute across different platforms. This puts a premium on portability and mobility of code, with some technical solutions arising as discussed in Section 3.5. Similar issues arise in storage and networking. There are several flavors of object relational databases competing in the marketplace, but a strong impetus to define unifying standards. The Internet technologies are evolving to accommodate a wider range of applications (e.g. high-quality voice and video), perhaps eventually displacing the stovepipe telephone and video distribution networks.

6.4.2 Shared responsibility

At the application layer there is heterogeneity introduced by the necessary partitioning of the application across hosts and across multiple organizations.

Many distributed applications involve two or more end-user organizations (e.g. supply chain management and business-to-business e-commerce more generally). This means that all the roles discussed earlier (such as planning, deployment, provisioning, operation) may be a shared responsibility. In addition, such applications must typically compose with legacy applications within these organizations. The shared ownership and operational control introduces many practical difficulties.

Since coordinated decision making on issues like what platform to adopt is impractical[114], other approaches are necessary. One approach is to define common standards, and then to introduce appropriate conversions to maintain interoperability with different platforms and legacy applications[115] (e.g., XML as a common representation for business documents of all types). Another approach is an intermediary, who takes responsibility for the interoperability with each of the organizations (e.g., the common business-to-business e-commerce intermediary being formed by the automotive industry [Cov00]).

There are many other issues in shared responsibility other than format and protocol conversions. Complications arise in all aspects of provisioning and operations.

6.4.3 Distributed partitioning

If an application is distributed over hosts, the issue arises as to how to partition it and why. Merely distributing an application does not by itself enable additional functionality –anything that can be done in a distributed environment can also be done centrally. There are nevertheless some compelling reasons for networking an application: First, the performance and scalability of an application may be improved when multiple hosts execute concurrently. Second, a centralized application must be administered centrally, whereas the administration and control of a distributed application can be partitioned. This is crucial for applications that span organizational boundaries where close coordination is not feasible. An example is business-to-business e-commerce, where an application may run across many organizations. Third, the security of an application is affected by its distribution. For example, if an application must access critical data assets of two organizations, each can exercise tighter control if each maintains control over its own data. Similarly, distributed data management can address privacy and ownership issues.

6.5 An industrial revolution?

The previous picture of separate software developers is a considerable simplification of reality. As mentioned earlier, most new software is constructed on a base of existing software. There are two cases to consider. Most of the time, development focuses on modifying and extending an existing code base, for example a new release of an existing application. Another case is where units of software design are specifically conceived to be *reusable* in multiple software developments[116]. Of course, a platform has this property of reusability, but the reuse idea can be extended to applications [Fra90, Gaf89].

Reuse has many attractions. A major one is the amortization of cost over multiple uses[117], and another is the reduced development time. Reusable software is also higher quality, because software used in different contexts and used more will have more effective "testing and repair by trial".

The reuse of software is analogous to the idea of standard interchangeable parts in the industrial revolution. Can software make a similar transition from handcrafted and individualized products to the composition of reusable elements licensed from elsewhere, as in the material world? Could it even move to the self-composition by users, who are then able to customize an application to their individual needs? While this has happened to some extent, and is clearly a goal for the future, this “industrial revolution in software” is much more difficult than in the material world.

Reusability can focus on two levels of design: architecture and individual modules. In software, the former is called a framework, and the latter a component. In both cases, the target of reuse is a *narrowed* range of applications[118], not *all* applications, since by definition infrastructure software targets reuse opportunities in the latter case.

Superdistribution [Cox96] is an example of how reusable components can be distributed quickly and widely while retaining economic incentives for the owner. Cox asserts that this helps curb complexity; as money is saved by using fewer components and thus it is economically favored to engineer streamlined lower-complexity solutions.

6.5.1 Frameworks

A *framework* is essentially an architecture that is reusable for multiple applications[119]. Thus, in essence it is a pre-plan for the decomposition of an application, including interface specifications. A framework can be customized by substituting different functionality in constituent modules, and extended by adding additional modules through defined gateways. Due to the wide variation in application requirements, the scope of a framework is necessarily limited: No single architecture will be suitable for a wide range of applications.

6.5.2 Components

One form of reuse is sharing, as happens with the infrastructure[120]. Another form of reuse is to incorporate preexisting modules into a software development. Roughly speaking, *software components* [Szy98] are reusable modules suitable for composition into multiple applications. Although the software community has seen many technologies, methodologies, and processes aimed at reuse, the consensus today is that component software is the most promising approach.

A component is designed to have minimal context dependencies. Instead of assuming many other modules to be present, a component provides connection points that allow it to be configured for a particular context[121]. Components can retain their identifiable individuality in a deployed application, allowing the application to be updated and extended by replacing or adding components. In contrast, current applications deploy a collection of executable or dynamically loadable modules that have configuration and context details “hard-wired in” and cannot be updated without being replaced as a whole.

Component-based architectures should be modular, particularly with respect to weak coupling of components, which eases their independent development and composability. Strong cohesion within components is less important, as components can themselves be hierarchically decomposed for purposes of implementation[122].

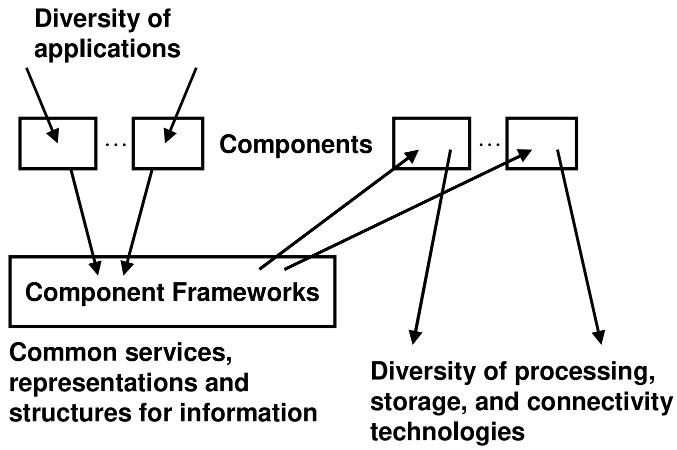

Figure 7. Component frameworks to separate dimensions of evolution.

While component-based systems are necessarily modular, components are reusable in different contexts, which is not a requirement of all modules. Component-based systems are much more difficult to design than modular ones. By relaxing the contextual assumptions a component can rely on, degrees of freedom are opened that lead to a potentially vast and open combinatorial space. Even theoretically, the set of possible configurations cannot be bounded, as the set of components is open and growing. Retaining quality under such conditions is a significant challenge that has yet to be fully addressed. Consequently, the market for software components started budding only around the mid 90's. The dominating practice still is more a craft (acquiring almost-right modules and modifying them to fit) rather than engineering (acquiring components designed to fit without modification in the context of a reference model).

The act of choosing, configuring and composing components is called *component assembly*. Constructing an application from components by configuring them all against one another, called a *peer-to-peer architecture*, does not scale beyond simple configurations because of the combinatorial explosion of created dependencies, all of which may need to be managed during the application evolution. A component framework can be used to bundle all relevant component connections and partial configurations, hierarchically creating a coarser-grain module[123]. Figure 7 illustrates how a framework can decouple disjoint dimensions[124]. An example of a component framework is an operating system, accepting device driver components to allow progress "below" and accepting application components to allow progress "above".

Allowing component frameworks to be components themselves creates a hierarchy. For example, an OS-hosted application can be turned into a framework by accepting plug-ins (which is a synonym for components). Although it may appear that component frameworks are layers as described in Section 6.3, the situation is more complex since component frameworks (unlike traditional layers) actively call components "layered" above them. Component frameworks are a recursive generalization of the idea of separating applications from infrastructure.

While most software systems today are not assembled predominantly from components, many hardware systems are. Adding customized software often differentiates them. Presently emerging component software technologies and methods may well trigger an industrial revolution in software in the future, where components purchased from outside firms are commonly composed into software. Customized modules (or components internally developed) can—and most likely will—still be used for differentiation.

7 Economic perspective

Microeconomics offers a number of insights into the business relationships and strategy of the software industry. This section builds on the earlier discussion to make some observations on the economic properties of software. However, a primary purpose of this paper is to stimulate more research into the economic and business characteristics of software; the discussion in this section is thus by no means complete or definitive.

Due to its low replication and distribution costs, and the fact that replicas of software are non-rival in use, traditional supply-demand relations typical of material goods do not apply. Thus, like information economics, software economics focuses on various issues relating to the incentives for investment in software development and ways in which suppliers can derive economic value from those investments.

7.1 Demand

Beyond the generic characteristics discussed in Section 2, software can serve as an intermediary for information access and add value in many other ways. Further advances, such as automated agents that perform increasingly sophisticated tasks on behalf of users and enhanced input-output based on speech and three-dimensional virtual reality, suggest that the value of software will continue to increase with time. The Internet has already resulted in remarkable new capabilities; overall, this revolution is still in its early stages.

7.1.1 Network effects vs. software category

Network effects have a strong influence on the software industry. However, there are considerable differences among different types of software.

Considering first the infrastructure, before the Internet different platforms could co-exist and compete for the same customers. The primary source of value related to market share was success in attracting application developers, i.e. secondary network effects. The platforms tended to segment the market—mainframes for back-office business applications, the PC's for personal productivity, and UNIX servers and workstations for the scientific and technical market and for departmental business applications. Post Internet, the platform for distributed applications becomes collectively all computer platforms, the network, and potentially expanding middleware (the middle layers in Figure 5). Virtually all infrastructure suppliers need to consider prominently their role in an ecosystem of complementary as well as competitive suppliers.

Prospective infrastructure solutions now face two related forms of network effects. First, they must achieve a sufficiently large user community, increasing the value they offer to each member of that community. Second, there is the "cart and horse" obstacle that infrastructure solutions offer value to end users only to the extent that they attract a significant base of applications layered on them, but application developers are only interested in layering on infrastructure with a significant penetration.

These two obstacles are difficult to overcome[125], but there are a couple of paths. Mirroring how the Internet was established, infrastructure solutions may initially arise in the research community, where they can attract experimental applications, and eventually the infrastructure and applications move together into the commercial marketplace. Otherwise, compelling capabilities might initially be bundled as a part of a successful application and subsequently spun off as a separate infrastructure product category[126].

Some categories of applications also experience considerable impediments due to network effects. Most distributed applications today follow the client-server model, in part because this model experiences less strong network effects—the first client of a new server application derives full value[127]. This may contribute to the success of the ASP model of application provisioning—the obstacle of getting the requisite software installed on many clients is overcome. On the other hand, applications that depend on many clients interoperating directly (called the peer-to-peer model) encounter stronger network effects. Examples include video conferencing, facsimile, and instant messaging. Nevertheless, there have been significant successes primarily because of the relative ease of distributing the necessary software over the network[128].

7.1.2 Lock-in

Customers often experience considerable *switching costs* in moving from one product to another, and this adds an impediment to competitive suppliers trying to attract customers [Sha99]. A complete application is composed of and depends on a number of complementary products, including different application components and infrastructure equipment and software. There are also less tangible complementary investments, such as the training of workers and the costs of changing the administration and operation of the software. Moving to a new software vendor usually involves switching costs associated with replacing complementary assets, retraining, etc. Lock-in attaches a negative value to a competitor's product equal to the switching costs, adding another barrier for that competitor to overcome[129].

Open standards are attractive to customers because they allow the mixing and matching of products from different vendors[130] and reduce switching costs. However, this only touches the surface. Lock-in is especially significant in business applications that have associated organizational processes and structures. If moving to a new vendor requires the reengineering of processes, reorganization and training of workers, and the disruption of business during deployment, the switching costs can be extraordinary. Competitive suppliers can overcome lock-in by subsidizing the customer's switching costs. Since that new supplier has already relinquished the incremental lock-in asset value, lock-in favors the supplier that initially acquires the customer[131].

In infrastructure, layering significantly reduces switching costs, since new infrastructure capabilities can be added without abandoning the old. This further explains the attractiveness of layering as a means of advancing the infrastructure. In fact, layering is a natural outcome of market forces in both the material and immaterial worlds.

7.2 Supply

On the supply side, software has similar characteristics as information. For example, the fixed creation costs are high, but manufacturing and distribution costs are very low, creating large supply economies of scale. Other costs, such as marketing and support, do not scale as well. The overall economies of scale allow a dominant supplier to undercut a competitor's pricing if necessary, resulting in positive feedback and further increasing market share. Scale economies [Sil87] also disallow a market approaching ”pure competition”[132] [Mar90], making it critically important for suppliers to differentiate their products from the competitors'.

7.2.1 Risk

The large creation costs of traditional software are largely sunk[133]. Application software is an experience good (it has to be experienced to be appreciated), increasing the difficulty of gaining adoptions. One way to overcome this is to take advantage of the low replication and distribution costs to offer a free trial. However, there is a limit to the attention of the users—ever more free trial offerings make this approach increasingly tedious. Thus, as a practical matter it is important to use a portfolio diversification strategy, investing in multiple products with overlapping product lifetimes, to mitigate risk.

7.2.2 Reusability

Advances in reusable software would have a substantial impact on software suppliers, reducing development costs and time and the attendant risk. Unfortunately, as discussed in Section 3.4, this reusability is considerably more difficult to achieve than in the material world due to the character of software. Nevertheless, viable component technologies have been emerging starting with the introduction of Microsoft COM as part of OLE 2 in 1992. Several fast-growing markets now exist [IDC99] and a number of companies have formed to fill the need for merchant, broker, and triage roles[134].

One significant open problem is trust and risk management: when software is assembled from components purchased form external suppliers, then warranty and insurance models are required to mitigate the risk of exposure. Due to the complexity and peculiarity of software, traditional warranty, liability laws, and insurance require rethinking in the context of software, an issue as important as the technical challenges.

7.2.3 Competition

The best way to derive revenue from software is through maximizing the value to the customer together with differentiation from competitor's products. However, it is difficult to prevent competitors from copying an application once it is released and available. Relative to some other industries (such as biotechnology) patents are relatively easy to circumvent by merely accomplishing the same ends another way. Copyrights have proven ineffective at preventing the copying of the features and "look and feel" of applications. Reproducing the same features and specifications independently in a "clean room" environment can circumvent copyrights.

What, then, are the fundamental deterrents to competitors, aside from intellectual property protections? There are several. First, enlightened competitors attempt to differentiate rather than copy, because they know that profits are difficult to obtain when there are undifferentiated competitors with substantial supply economies of scale. Second, a supplier, who has achieved significant market share and economies of scale, can employ *limit pricing*, which takes into account the high creation costs faced by a new entrant [Gas71]. Third, lock-in of customers is advantageous to the supplier with the largest market share. Switching costs may require a competitor to subsidize a customer's switch, either explicitly or through price discounting. Suppliers thus attempt to maximize the lock-in of customers, for example by adding proprietary features or enhancing applications interoperability with complementary products. On the other hand, there are numerous strategies competitors can employ to reduce the customer's switching costs, such as offering translations or backward compatibility [Sha99], many of these measures specific to software. This is a promising area for research.

7.2.4 Dynamic Supply Chains

For material goods, suppliers and customers can have long-term contractual relationships, or procurement can occur dynamically in a marketplace (electronic or otherwise). For software, these means are possible, but in addition supply chains can be made to be fully "self-aware", as illustrated by superdistribution [Cox96]. Components can be freely exchanged, even directly among customers, and the components themselves can initiate and enforce a fair compensating monetary flow using an enabling micro-billing infrastructure.

7.2.5 Rapidly expanding markets

Software markets are often very rapidly expanding, especially in light of the low replication and distribution costs and distribution delay. This engenders special challenges and strategies for suppliers. For example, the simple theory of lock-in does not apply in a rapid-growth situation, as capturing a significant fraction of the customer base does not, by itself, suffice to gain a longer-term advantage. Rather, for as long as growth is rapid and network effects are not strong, new competitors can enter and attract a large part of new customers entering the market—for example, by aiming at a more attractive price-value point, offering a more attractive bundle, or offering other advantages such as better integration.

Thus, initial strength in new technologies—and software is no exception—grants a first-mover advantage, but not necessarily a lock-in advantage. Leveraging the first- (or early-) mover advantage requires rapid and continuing innovation, despite a lock-in advantage with the initial but rapidly marginalized customer base.

7.3 Pricing

Pricing is an important issue in any supplier-customer relationship. The large supply economies of scale make pricing a particular challenge for software suppliers. Unit costs offer little guidance, since they depend strongly on unit sales, opening the space for a wide variety of pricing strategies with an equally wide array of ramifications.

There are several dimensions. First, does an identical pricing schedule apply to all customers, or is there price discrimination? Forms of price discrimination include basing price on individual customers' value or willingness to pay, segmentation of the customer population, or versioning, in which customers self-select the most attractive price-quality tradeoff from among several alternatives. Second, is the price usage dependent? Usage-based pricing requires monitoring and billing of usage, or this can be avoided with crude usage metrics such as the allowable peak number of individual users, the number of servers the software is installed on, or the number of clients able to access the software. Third, what are the terms and conditions of the sale? Some common options include paying once, with the right to use indefinitely, leasing for a limited period, paying per use, or subscription arrangements involving periodic payments. Another class of terms is a warrantee on correct functioning or on performance attributes (the latter is called a *service level agreement* [Hil93, Koc98]). Fourth, what is bundled into the sale? Beyond the software itself, options include maintenance upgrades, customer support, and new releases. In the case of an ASP, provisioning and operations are also bundled. Fifth, who pays? In some cases, it is not the end-user but a third party like an advertiser or a cross subsidy from other products.

These dimensions of pricing are sometimes coupled. For example, usage-based or per-use pricing inherently requires periodic billing, a prepayment and debit system, or a pay-before-use infrastructure (such as digital cash). A supplier shouldn't promise new releases (with the attendant ongoing development costs) unless they are purchased separately or there is ongoing subscription revenue. Similarly an ASP should expect subscription revenues to offset operational costs.

Finally, all ways of selling software, including bundling with provisioning and operations or not, offer a full range of pricing options. While licensing and customer-installation is usually associated with fixed pricing, with networked computers it would be possible to use subscription or per-transaction pricing. Software sold as an application service is usually associated with subscription pricing or coverage by third-party advertising, but could be sold at a fixed price.

7.3.1 Value pricing and versioning

For products well differentiated from the competition, the best supplier strategy is *value pricing*: base prices on the customer willingness to pay [Sha99]. Value pricing is, however, complicated by the wide dispersion in willingness to pay among customers. To maximize revenue, value pricing requires price discrimination.

There are a number of different price discrimination strategies. One particularly applicable to software is *versioning*. A portfolio of products is created that offers different levels of features, quality, and performance. With appropriate pricing, customers will self-select based on their willingness to pay. Frequently, a minimal version is offered for free, for a limited-time trial or indefinitely. This bears little marginal cost to the supplier, and serves to familiarize the customer with the product in the hope of upgrading them to a paying version or inducing them or others to purchase a complementary product. An example of the latter is the Web, where the browser is free and the server with complementary value-added features is sold.

Some business models offer more flexibility for price discrimination than others. Fixed pricing of "shrink wrapped" software offers minimal opportunity for price discrimination. At the other extreme, an individually negotiated license for a custom-developed application can take into account attributes like usage and impact discussed in Section 2. An attractiveness of the ASP model is some flexibility to base pricing on willingness to pay, usage, and context.

7.3.2 Variable pricing

The user's willingness to pay depends on the many contributors to value described earlier. Some of these, such as usage and quality, can actually be different at different times, even for the same user. This suggests forms of variable pricing in which the price is dependent on patterns of use.

Actually implementing variable pricing is a challenge. An example is pricing based on usage. While direct monitoring of usage (time or transactions) is rare, suppliers frequently approximate usage by basing prices on the number of "seats" where an application is available or the number of computers on which the software is installed, irrespective of actual usage. A variant is the "floating license", in which pricing is based on the peak number of concurrent users without regard to the number of seats or users. With the ubiquity of the Internet, it becomes possible to monitor usage more directly, and similarly the ASP model naturally admits the monitoring of use. A fundamentally different approach is to base pricing on direct impact rather than usage, such as per-transaction pricing for ASP e-commerce intermediaries.

The business model and pricing strategy can have a significant impact on supplier incentives and targets for investment. For example, usage-based pricing provides an ongoing revenue stream even without releases, and thus reduces the pressure to continually add functionality (unless of course it is targeted at increased usage).

7.3.3 Bundling

As with other goods, bundling of products is another way to mitigate the dispersion of customer demand [Sha99]. That dispersion is often lower for a bundle than for its constituents, making pricing simpler and ultimately increasing total revenues. In software, making the constituents complementary and composable can enhance the value of a bundle[135].

7.3.4 Third party revenue

With low marginal costs, advertising is a viable mechanism to derive revenues from a third party rather than users. This is particularly true for the ASP subscription model, since there is an opportunity to push a stream of targeted advertisements, and increasing their effectiveness by taking into account the application context. In this case, the determinant is the value of the attention of the user to the advertiser, rather than the direct value to the user. Unlike traditional media, advertisements in networked media can be updated dynamically and usually include hyperlinks to the advertiser's Web site, where unlimited information can be offered.

7.4 Evolution

To reduce the risk of competitor entry and create a stream of revenues, it is typical to create a moving target by offering a stream of new releases, each with improved features, quality, and performance. Maintenance upgrades may be offered for free[136], but new releases can be sold.

Releases also help support maintenance and improvements that attract new customers, create new versions, and deter potential competitors. Once a significant market share is held, the greatest competition for each new release is the installed base of older releases.

Although immaterial software does not "wear out" like material goods, absent upgrade it does inevitably deteriorate over time in the sense that changing user requirements and changes to complementary products render it less suitable. Thus, software demands new releases as long as there is a viable user base. Upgrades entail some risks: a new release may discontinue support for older data representations (alienating some customers), or may suddenly fail to interoperate with complementary software.

Legacy software may eventually become a financial burden to the supplier. The user community may dwindle to the point that revenues do not justify the investment in releases, or the supplier may want to replace the software with an innovative new version. Unfortunately, terminating investments in new releases will alienate existing users by stranding them with deteriorating (and eventually unusable) software. Thus, the installed base sometimes becomes an increasing burden. Components offer a smoother transition strategy provided old and new versions of a component can be installed side-by-side[137]. Then, old versions can be phased out slowly, without forcing clients of that old version to move precipitously to the new version.

7.5 Complementarity

Similarly to other markets, it is common to offer a portfolio of complementary products. This reduces risk, sales and marketing costs, and offers the consumer systems integration and a single point of contact for sales and support.

Nevertheless, software suppliers depend heavily on complementary products from other suppliers, particularly through layering. Each supplier wants to differentiate its own products and minimize its competition, but conversely desires strong competition among its complementers so that its customers enjoy overall price and quality advantages.

8 The future

More so than most technologies and markets, software has been in a constant state of flux. The merging of communications with storage and processing represents a major maturation of the technology—there are no remaining major gaps. However, the market implications of this technological step have only begun to be felt. In addition, there are a few other technological and market trends that are easily anticipated, because they have already begun. While we don't attempt to predict their full implications, we now point out what they are and some of their implications.

8.1 Information appliances

Instead of installing specialized applications on general computers, software can be bundled with hardware to focus on a narrower purpose. *Information appliances* take software applications co-existing in the PC, and bundle and sell them with dedicated hardware[138]. This exploits the decreasing cost of hardware to create devices that are more portable, ergonomic, and with enhanced usability[139].

Software in the appliance domain assumes characteristics closer to traditional industrial products. Both the opportunities and technical challenges of composability are largely negated. In most instances, maintenance and upgrade become a step within the appliance product activity, rather than a separable software-only process[140].

8.2 Pervasive computing

Another trend is embedding software-mediated capabilities within a variety of existing material products[141]. The logical extension of this is information technology (including networked connectivity as well as processing and storage) embedded in most everyday objects, which is termed *pervasive computing* [Mak99, Cia00]. The emphasis is different from information appliances, in that the goal is to add capability and functionality to the material objects around us—including many opportunities that arise when these objects can communicate and coordinate—as opposed to shipping existing capabilities in a new form. Our everyday environment becomes a configurable and flexible mesh of (largely hidden from view) communicating and computing nodes that take care of information processing needs less explicitly expressed and deriving from normal activities.

Pervasive computing takes software in the opposite direction from information appliances. Composability that is flexible and opportunistic and almost universal becomes the goal. This is a severe technical challenge[142]. Further, taking advantage of information technology to increase the complementarity of many products in the material world becomes a new and challenging goal for product marketing and design.

8.3 Mobile and nomadic information technology

Many users have accessed the Internet from a single access point. Increasingly, *nomadic* users connect the network from different access points (as when they use laptops while traveling). There are two cases: the appliance or computer itself can be relocated or the user can move from one appliance or computer to another. The advent of wireless Internet access allows *mobility*: users change their access point even while they are using an application[143].

Maintaining ideal transparency to the nomadic or mobile user raises many new challenges for software engineers and managers. The infrastructure should recreate a consistent user environment wherever the user may arise or move (including different access points or different appliances or computers). Either, applications need to be more cognizant of and adjust to wide variations in communications connectivity, or the infrastructure needs to perform appropriate translations on behalf of the applications[144]. A severe challenge is achieving all this when necessarily operating over distinct ownership and administrative domains in a global network.

8.4 A component marketplace

The assembly of applications from finer-grained software components is very limited as an internal strategy for individual software suppliers, because their uses may not justify the added development costs[145]. Like infrastructure software, the full potential unfolds only with the emergence of a marketplace for components.

However, such markets may well turn out to be very different from hardware and material goods, which are typically sold for a fixed price. One difference has already been discussed: software components are protected by intellectual property rather than title, and will typically be licensed rather than sold. More fundamentally, the pricing may be much more variable, potentially including a number of factors including usage. The possibilities are endless, and this is a useful area of investigation.

8.5 Pricing and business models

The rising popularity of the ASP model for provisioning and operations demonstrates the changing business model of the software industry. There are several factors driving this. First, the increasing ubiquity of high-performance networks opens up new possibilities. Second, as the assembly of finer-grained software components replaces monolithic applications, and new value is created by the composition of applications, a given application may actually have multiple vendors. Pervasive computing pushes this trend to the extreme, as the goal is to allow composition of higher-level capabilities from different computing devices, often from different vendors[146]. Third, mobility creates an endless variety of possible scenarios for the partitioned ownership and operation of the supporting infrastructure. Fourth, the obstacles of provisioning and operating applications become much more daunting to the user in a world with much greater application diversity, and applications composed from multiple components.

Traditional usage-independent pricing models based on hosts or users supported become less appropriate in all these scenarios. Instead, pricing should move to usage-based subscription models. Such pricing models require infrastructure support for transparent, efficient, and auditable billing against delivered services.

For example, components may be sold for subscription (usage is monitored, and micro-payments flow to component vendors). Since most components introduce a miniature platform, which other components can build on, this encourages widespread component adoption, and minimizes the initial barriers to entry where earlier component offerings are already established.

While the details are unclear, the changing technology is stimulating widespread changes in industrial organization and business models.

9 Conclusions

While individually familiar, software brings unusual combinations of characteristics on the supply and demand sides. Its unparalleled flexibility, variability, and richness are countered by equally unparalleled societal, organizational, technical, financial, and economic challenges. Due to these factors—continually rocked by unremitting technological change—today's software marketplace can be considered immature.

We assert that there are substantial opportunities to understand better the challenges and opportunities of investing in, developing, marketing, and selling software, and to use this understanding to conceptualize better strategies for the evolution of software technology as well as business models that better serve suppliers and customers. It is hoped that this paper takes a first step toward realization of this vision by summarizing our current limited state of understanding.

Software is subject to a foundation of laws similar to (and sometimes governed by) the laws of physics, including fundamental theories of information, computability, and communication. In practical terms these laws are hardly limiting at all, especially in light of remarkable advances in electronics and photonics that will continue for some time. Like that other immaterial good that requires a technological support infrastructure, information, software has unprecedented versatility: the only really important limit is our own imagination. That, plus the immaturity of the technology and its markets, virtually guarantees that this paper has not captured the possibilities beyond a limited vision based on what is obvious or predictable today. The possibilities are vast, and largely unknowable.

While the wealth of understanding developed for other goods and services certainly offer many useful insights, we feel that the fundamentals of software economics are yet to be conceptualized. Competitive market mechanisms, valuation and pricing models, investment recovery, risk management, insurance models, value chains, and many other issues should be reconsidered from first principles to do full justice to this unique good.

References

[Bak79] Baker, Albert L.; Zweben, Stuart H. "The Use of Software Science in Evaluating Modularity Concepts". *IEEE Transactions on Software Engineering*, March 1979, SE-5(2): 110-120.

[Bal97] Baldwin, Carliss Y; Clark, Kim B. "Managing in an age of modularity". *Harvard Business Review*, Sep/Oct 1997, 75(5): 84-93.

[BCK98] Bass, Len; Clements, Paul; Kazman, Rick. *Software Architecture in Practice*, Addison-Wesley, 1998.

[Boe00] Boehm, B; Sullivan, K. "Software Economics: A Roadmap," in *The Future of Software Engineering*, special volume, A. Finkelstein, Ed., *22nd International Conference on Software Engineering*, June, 2000.

[Boe81] Boehm, B.. *Software Engineering Economics***,** Englewood Cliffs, N.J. : Prentice-Hall, 1981.

[Boe84] Boehm, Barry W. "Software Engineering Economics". *IEEE Transactions on Software Engineering*, Jan 1984, SE-10(1): 4-21.

[Boe99] Boehm, B; Sullivan, K. "Software economics: Status and prospects". *Information & Software Technology*, Nov 15, 1999, 41(14): 937-946.

[Bos00] Bosch, Jan. *Design and Use of Software Architectures*, Addison Wesley, 2000

[Bro98] Brown, William J.; Malveau, Raphael C.; Brown, William H.; McCormick,; Hays W., III; Mowbray, Thomas J. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley & Sons, 1998.

[Bul00] Bulkeley, William M. "Ozzie to unveil Napster-style networking" *Wall Street Journal Interactive Edition*, October 24. (http://www.zdnet.com/zdnn/stories/news/0,4586,2644020,00.html)

[Chu92] Church, Jeffrey; Gandal, Neil. "Network Effects, Software Provision, and Standardization". *Journal of Industrial Economics*, Mar 1992, 40(1): 85-103.

[Cia00] Ciarletta, L.P., Dima, A.A. "A Conceptual Model for Pervasive Computing", Workshop on Pervasive Computing; in: *Proceedings of the 29th International Conference on Parallel Computing 2000*, Toronto, Canada, 21-24 Aug 2000.

[Cla93] Clark, J R; Levy, Leon S. "Software economics: An application of price theory to the development of expert systems" *Journal of Applied Business Research*, Spring 1993, 9(2): 14-18.

[Com96] Compaq. "White paper: How DIGITAL FX!32 works". (http://www.support.compaq.com/amt/fx32/fx-white.html.)

[Cov00] Covisint "Covisint Establishes Corporate Entity – Automotive e-business exchange becomes LLC", December 2000. (http://www.covisint.com/)

[Cox96] Cox, B. *Superdistribution: Objects as Property on the Electronic Frontier*; Addison Wesley 1996. (http://www.virtualschool.edu/mon)

[CSA98] Workshop report, *OMG DARPA Workshop on Compositional Software Architectures*, February 1998.
(http://www.objs.com/workshops/ws9801/report.html)

[Dav90] David, Paul A., and Shane Greenstein. 1990. "The Economics of Compatibility Standards: An Introduction to Recent Research," *Economics of Innovation and New Technology* 1(1-2): 3-41.

[DFS98] Devanbu, P.; Fong, P.; Stubblebine, S. "Techniques for trusted software engineering" In: *Proceedings of the 20th International Conference on Software Engineering* (ICSE'98), Kyoto, Japan, April 1998.
(http://seclab.cs.ucdavis.edu/~devanbu/icse98.ps)

[Fra90] Frakes, W. B.; Gandel, P. B. "Representing Reusable Software". *Information & Software Technology*, Dec 1990, 32(10): 653-664.

[Gaf89] Gaffney, J. E., Jr.; Durek, T. A. "Software Reuse - Key to Enhanced Productivity: Some Quantitative Models". *Information & Software Technology*, Jun 1989, 31(5): 258-267.

[Gas71] Gaskins. "Dynamic Limit Pricing: Optimal Pricing Under Threat of Entry", *J. Econ. Theory* 306, 1971.

[Goe9?] Ben Goertzel, "The Internet Economy as a Complex System", 199?. (http://www.goertzel.org/ben/ecommerce.html)

[Gul93] *Analytical methods in software engineering economics*. Thomas R. Gulledge, William P. Hutzler, eds. Berlin ; New York : Springer-Verlag, c1993.

[Hil93] Hiles, A. *Service Level Agreements – Managing Cost and Quality in Service Relationships*, Chapman & Hall, London, 1993.

[How97] Howard, J.D. "An Analysis Of Security Incidents On The Internet", *PhD thesis*, Carnegie Mellon University, Pittsburgh, PA, April 1997. (http://www.cert.org/research/JHThesis/Start.html)

[IDC99] Steve Garone and Sally Cusack. "Components, objects, and development environments: 1999 worldwide markets and trends". International Data Corporation, June 1999.

[Jun99] Jung, Ho-Won; Choi, Byoungju. "Optimization models for quality and cost of modular software systems". *European Journal of Operational Research*, Feb 1, 1999, 112(3): 613-619.

[Kan89] Kang, K. C.; Levy, L. S. "Software Methodology in the Harsh Light of Economics". *Information & Software Technology*, Jun 1989, 31(5): 239-250.

[Kat85] Katz, Michael, and Carl Shapiro. 1985. "Network Externalities, Competition, and Compatibility," *American Economic Review* 75(3): 424-440.

[Kat86] Katz, Michael L.; Shapiro, Carl. "Technology Adoption in the Presence of Network Externalities". *Journal of Political Economy*, Aug 1986, 94(4): 822-841.

[Ken98] Kemerer, Chris F. "Progress, obstacles, and opportunities in software engineering economics". *Communications of the ACM*, Aug 1998, 41(8): 63-66.

[Koc98] Koch, Christopher. "Service level agreements: put IT in writing", *CIO Magazine*, 15 Nov 1998. (http://www.cio.com/archive/111598_sla.html)

[Lan00] Langlois, Richard, "Modularity in Technology and Organization", to appear in the *Journal of Economic Behavior and Organization*.

[Lan92] Langlois, Richard N. "External economies and economic progress: The case of the microcomputer industry". *Business History Review*, Spring 1992, 66(1): 1-50.

[Lan92] Langlois, Richard N.; Robertson, Paul L. "Networks and Innovation in a Modular System: Lessons from the Microcomputer and Stereo Component Industries". *Research Policy*, Aug 1992, 21(4): 297-313.

[Lev87] Levy, L. S., *Taming the Tiger: Software Engineering and Software Economics*, Springer-Verlag, Berlin, FRG, 1987.

[Lew97] Ted Lewis, *Friction-Free Economy*. HarperBusiness, 1997. (http://www.friction-free-economy.com/)

[LL96] Lee, Peter; Leone, Mark, "Optimizing ML with run-time code generation", *ACM SIGPLAN Notices*, 1996, 31(5): 137-148.

[Mak99] Makulowich, John. "Pervasive Computing: 'The Next Big Thing'" *Washington Technology Online*, 19 July 1999. (http://www.wtonline.com/vol14_no8/cover/652-1.html)

[Mar90] Marshall, Alfred. *Principles of Economics*, first edition: 1890. Reprinted in Great Minds Series, Prometheus Books, 1997.

[Mes99a] David G. Messerschmitt, *Understanding Networked Applications: A First Course*. Morgan Kaufmann, 1999.

[Mes99b] David G. Messerschmitt, *Networked Applications: A Guide to the New Computing Infrastructure*. Morgan Kaufmann, 1999.

[Mes99c] D.G. Messerschmitt, "The Prospects for Computing-Communications Convergence". *Proceedings of MÜNCHNER KREIS, Conference "VISION 21:Perspectives for the Information and Communication Technology"*, Munich Germany, Nov. 25, 1999. (http://www.EECS.Berkeley.EDU/~messer/PAPERS/99/Munich.PDF)

[Net] Nepliance, Inc. (http://www.netpliance.com/iopener/)

[Nie00] Nielsen, J. *Designing Web Usability: The Practice of Simplicity.* New Riders Publishing, Indianapolis, 2000.

[Nie93] Nielsen, J. "Noncommand user interfaces." *Communications of the ACM*, April 1993, 36(4): 83-99. (http://www.useit.com/papers/noncommand.html)

[Par72] Parnas, David L. 1972. "On the Criteria for Decomposing Systems into Modules," Communications of the ACM 15(12): 1053-1058 (December).

[Pfl97] Pfleeger, Charles P. *Security in Computing*, 2nd edition, Prentice Hall, 1997.

[Pre00] Pressman, Roger S. *Software Engineering: A Practitioner's Approach.* (Fifth Edition) McGraw-Hill, 2000.

[Rob95, Lan92] Robertson, Paul L; Langlois, Richard N. "Innovation, networks, and vertical integration". *Research Policy*, Jul 1995, 24(4): 543-562.

[Roy70] Royce, W.W. "Managing the development of large software systems", IEEE WESCON, August 1970.

[San96] Sanchez, Ron; Mahoney, Joseph T. "Modularity, flexibility, and knowledge management in product and organization design". Strategic Management Journal, Winter 1996, 1763-76.

[Sch88] Schattke, Rudolph. "Accounting for Computer Software: The Revenue Side of the Coin". *Journal of Accountancy*, Jan 1988, 165(1): 58-70.

[Sha99] Carl Shapiro and Hal R. Varian, *Information Rules: A Strategic Guide to the Network Economy*. Harvard Business School Press, 1999.

[Sil87] Silvestre, Joaquim. "Economies and Diseconomies of Scale", in: The New Palgrave: A Dictionary of Economics, ed. by John Eatwell, Murray Milgate, and Peter Newman. London: Macmillan, London, 1987, (2): 80-83.

[Sla98] Slaughter, Sandra A; Harter, Donald E; Krishnan, Mayuram S. "Evaluating the cost of software quality". *Communications of the ACM*, Aug 1998, 41(8): 67-73.

[SOT00] Suganuma, T.; Ogasawara, T.; Takeuchi, M.; Yasue, T.; Kawahito, M.; Ishizaki, K.; Komatsu, H.; Nakatani, T. "Overview of the IBM Java just-in-time compiler", *IBM Systems Journal*, 2000, 39(1): 175-193.

[Sul99] Sullivan, Jennifer. "Napster: Music Is for Sharing", Wired News, 1 November 1999. (http://www.wired.com/news/print/0,1294,32151,00.html)

[Sun99] "The Java Hotspot™ performance engine architecture – A white paper about Sun's second generation performance technology", April 1999. (http://java.sun.com/products/hotspot/whitepaper.html)

[Szy98] Clemens Szyperski, *Component Software—Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[The84] Thebaut, S. M.; Shen, V. Y. "An Analytic Resource Model for Large-Scale Software Development". *Information Processing & Management*, 1984, 20(1/2): 293-315.

[Tor98] Torrisi, S., *Industrial Organization and Innovation : An International Study of the Software Industry*. Edward Elgar Pub, 1998.

[UPA] Usability Professionals' Association (http://www.upassoc.org/)

[Upt92] Upton, David M. "A flexible structure for computer-controlled manufacturing systems", Manufacturing Review, 1992, 5 (1): 58-74. (http://www.people.hbs.edu/dupton/papers/organic/WorkingPaper.html)

[Vac93] Vacca, John. "Tapping a gold mine of software assets". *Software Magazine*, Nov 1993, 13(16): 57-67.

[Ver91] *The Economics of information systems and software*. Richard Veryard, ed. Oxford ; Boston : Butterworth-Heinemann, 1991.

[W3C95] World Wide Web Consortium. "A Little History of the World Wide Web" (http://www.w3.org/History.html)

[W3CP] World Wide Web Privacy. (http://www.w3.org/Privacy/)

[War00] Ward, Eric, "Viral marketing involves serendipity, not planning". *B to B*, Jul 17, 2000, 85(10): 26.

The authors

David G. Messerschmitt is the Roger A. Strauch Professor of Electrical Engineering and Computer Sciences at the University of California at Berkeley. From 1993-96 he served as Chair of EECS, and prior to 1977 he was with AT&T Bell Laboratories in Holmdel, N.J. Current research interests include the future of wireless networks, the economics of networks and software, and the interdependence of business and technology. He is active in developing new courses on information technology in business and information science programs, introducing relevant economics and business concepts into the computer science and engineering curriculum, and is the author of a recent textbook, *Understanding Networked Applications: A First Course*. He is a co-founder and former Director of TCSI Corporation. He is on the Advisory Board of the Fisher Center for Management & Information Technology in the Haas School of Business, the Directorate for Computer and Information Sciences and Engineering at the National Science Foundation, and recently co-chaired a National Research Council study on the future of information technology research. He received a B.S. degree from the University of Colorado, and an M.S. and Ph.D. from the University of Michigan. He is a Fellow of the IEEE, a Member of the National Academy of Engineering, and a recipient of the IEEE Alexander Graham Bell Medal.

Clemens A. Szyperski is a Software Architect in the Component Applications Group of Microsoft Research, where he furthers the principles, technologies, and methods supporting component software. He is the author of the award-winning book *Component Software: Beyond Object-Oriented Programming* and numerous other publications. He is the charter editor of the Addison-Wesley *Component Software* professional book series. He is a frequent speaker, panelist, and committee member at international conferences and events, both academic and industrial. He received his first degree in Electrical Engineering in 1987 from the Aachen Institute of Technology in Germany. He received his Ph.D. in Computer Science in 1992 from the Swiss Federal Institute of Technology (ETH) in Zurich under the guidance of Niklaus Wirth. In 1992-93, he held a postdoctoral scholarship at the International Computer Science Institute at the University of California, Berkeley. From 1994-99, he was tenured as associate professor at the Queensland University of Technology, Brisbane, Australia, where he still holds an adjunct professorship. In 1993, he co-founded Oberon microsystems, Inc., Zurich, Switzerland, with its 1998 spin-off, esmertec inc., also Zurich.

Endnotes

---

[1] In fact, often the term technology is defined as the application of physical laws to useful purposes. By this strict definition, software would not be a technology. However, since there is a certain interchangeability of software and hardware, as discussed momentarily, we do include software as a technology.

[2] The theoretical mutability of hardware and software was the original basis of software patents, as discussed in Section 5. If it is reasonable to allow hardware inventions to be patented, then it should be equally reasonable to allow those same inventions, but embodied by software, to be patented.

[3] The computer is arguably the first product that is fully programmable. Many earlier products had a degree of parameterizability (e.g. a drafting compass) and configurability (e.g. an erector set). Other products have the flexibility to accommodate different content (e.g. paper). No earlier product has such a wide range of functionality non-presupposed at the time of manufacture.

[4] The primary practical issues are complexity and performance. It is somewhat easier to achieve high complexity in software, but moving the same functionality to hardware improves performance. With advances in computer-aided design tools, hardware design has come to increasingly resemble software programming.

[5] By representation, we mean the information can be temporarily replaced by data and later recovered to its original form. Often, as in the sound and picture examples, this representation is only approximated. What is recovered from the data representation is an approximation of the original.

[6] The usage of these terms is sometimes variable and inconsistent. For example, the term data is also commonly applied to information that has been subject to minimum interpretation, such as acquired in a scientific experiment.

[7] Analog information processing--for example, in the form of analog audio and video recording and editing—remains widespread. Analog is being aggressively displaced by digital to open up opportunities for digital information processing.

[8] In reality, storage cannot work without a little communication (the bits need to flow to the storage medium) and communication cannot work without a little storage (the bits cannot be communicated in zero time).

[9]Note that the "roles" of interest to managers (such as programmers and systems administrators) have some commonality with the perspectives. The distinction is that the perspectives are typically more general and expansive.

[10] The perspectives chosen reflect the intended readership of this paper. We include them all because we believe they all are relevant and have mutual dependencies.

[11] As described in Section 4, this cycle typically repeats with each new software release.

[12] The difference between value and cost is called the *consumer surplus*. Software offering a larger consumer surplus is preferred by the consumer.

[13] Often, value can be quantified by financial metrics such as increased revenue or reduced costs.

[14] Of course, if the greater time spent reflects poor design, greater usage may reflect lower efficiency and thus represents lower value.

[15] "Observed" is an important qualifier here. The actual number of defects may be either higher or lower than the observed one—it is higher than observed if some defects don't show under typical usage profiles; it is lower than observed if a perceived defect is actually not a defect but a misunderstanding on how something was supposed to work. The latter case could be re-interpreted as an actual defect in either the intuitiveness of the usage model, the help/training material, or the certification process used to determine whether a user is sufficiently qualified.

[16] For example, a slow activity can be masked by a multitude of attention-diverting faster activities.

[17] Performance is an important aspect of software composition (see Section 2.8): two separately fast components, when combined, can be very slow—a bit like two motors working against each other when coupled. The exact impact of composed components (and the applied composition mechanism) on overall performance is hard to predict precisely for today's complex software systems.

[18] Although this quality dilemma is faced by all engineering disciplines, many benefit from relatively slow change and long historical experience, allowing them to deliver close-to-perfect products. IT as well as user requirements are and have always changed rapidly, and any stabilization is accurately interpreted as a leading indicator of obsolescence.

[19] In theory, software could be tested under all operational conditions, so that flaws could be detected and repaired during development. While most flaws can be detected, the number of possible conditions in a complex application is so large as to preclude any possibility of exhaustive testing.

[20] Such modes might include mouse or keyboard, visual or audio, context-free or context-based operations.

[21] An example would be an initial "discovery" of features supported through several likely paths, while later repetitive use of certain features can be fine-tuned to minimize the required number of manipulative steps. Typical examples include the reconfiguration of user interface elements or the binding of common commands to command keys.

[22] Suitable mechanisms to support security or privacy policies can range from simple declarations or warnings at "entry points" to total physical containment and separation. For all but the most trivial degrees of resiliency, hardware and physical location support is required.

[23] Unfortunately, it is not well understood how to construct software that can meet changing needs. The best attempts add considerable ability to parameterize and configure, and attempt modular architectures, in which the user can mix and match different modules (see Section 3 for further discussion). As a practical matter, information systems are often a substantial obstacle to change.

[24] Relevant performance parameters are instruction rate (instructions per second), storage density (bits per unit area or per chip), and communications bitrate (bits per second).

[25] Thus far, reductions in feature size (which relates directly to improved speed at a given cost) by a fixed percentage tend to cost roughly the same, regardless of the absolute. Thus, like compound interest, the cumulative improvement is geometric with time (roughly 60% per year compounded).

[26] The Semiconductor Industry Association has developed a roadmap for semiconductor development over the next 6 years. This roadmap specifies the needed advances in every area, and serves to coordinate the many vendors who contribute to a given generation of technology.

[27] The inadequacy of computers even a few years old with today's applications illustrates concretely the importance of advancing technology to the software industry.

[28] Compilation is typically seen as yielding pre-checked efficient object code that lacks the flexibility of dynamic, on-demand modifiability and, importantly, the flexibility to execute on a variety of target machines with different execution models. Interpretation is typically seen as yielding a more lightweight and flexible model, but at the price of very late checking and reduced efficiency. Everyone has suffered from the late checking applied to interpreted code: a visited Web page "crashes" with an error message indicating some avoidable programming error in a script attached to the Web page. While early checking during compilation cannot (*ever!*) eliminate all errors, modern languages and compiler/analyzer technology have come quite far in eliminating large classes of errors (thus termed "avoidable" errors).

[29]This is illustrated by Java. A common (but not the only) approach is to compile Java source code into Java bytecode, which is an intermediate object code for an abstract execution target (the so-called Java virtual machine). This bytecode can then be executed on different targets by using a target-specific interpreter. If all checking happens in the first step and if the intermediate object code is efficiently mapable to native code, then the advantages of compilation and interpretation are combined. The software unit can be compiled to intermediate form, which can then be distributed to many different target platforms, each of which relies on interpretation to transform to the local physical execution model.

[30] Java is more than a language. It includes a platform, implemented on different operating systems, that aims at supporting full portability of software.

[31] By monitoring the performance, the online optimizer can dynamically optimize critical parts of the program. Based on usage profiling, an online optimizer can recompile critical parts of the software using optimization techniques that would be prohibitively expensive in terms of time and memory requirements when applied to all of the software. Since such a process can draw on actually observed system behavior at "use time", interpreters combined with online optimizing compilation technology can exceed the performance achieved by traditional (ahead-of-time) compilation.

[32] Java source code is compiled into Java bytecode—the intermediate object code proprietary to Java. Bytecode is then interpreted by a Java Virtual Machine (JVM). All current JVM implementations use just-in-time compilation, often combined with some form of online optimization, to achieve reasonable performance.

[33] There is nothing special about intermediate object code: one machine's native code can be another machine's intermediate object code. For example, Digital (now Compaq) developed a "Pentium virtual machine" called FX!32 [Com96] that ran on Digital Alpha processors. FX!32 used a combination of interpretation, just-in-time compilation, and profile-based online optimization to achieve impressive performance. At the time, several Windows applications, compiled to Pentium object code, ran faster on top of FX!32 on top of Alpha, than on their native Pentium targets.

[34] This approach uses a digital signature. Any form of verification of a vendor requires the assistance of a trusted authority, in this case called a certificate authority (CA). The CA provides the software vendor with a secret key that can be used to sign the code in a way that can be verified by the executing platform [Mes99a]. The signature does not limit what is in the code and thus has no impact on the choice of object code format. Microsoft's Authenticode technology uses this approach.

[35] Java bytecode and the .NET Framework intermediate language use this approach. A generalization of the checking approach is presently finding much attention: *proof-carrying code*. The idea is to add enough auxiliary information to an object code that a receiving platform can check that the code meets certain requirements. Such checking is, by construction, much cheaper than constructing the original proof: the auxiliary information guides the checker in finding a proof. If the checker finds a proof, then the validity of the proof rests only on the correctness of the checker itself, not on the trustworthiness of either the supplied code or the supplied auxiliary information. The only thing that needs to be trusted is the checker itself.

[36] The operating system is an example of infrastructure (as opposed to application) software (see Section 6).

[37] The stages up to and (in the extreme) including requirements need to consider the available code base to efficiently build on top of it.

[38]Traditionally, the two most important tools of a software developer were source code editors and compilers. With the availability of integrated development environments, the toolkit has grown substantially to include functional and performance debuggers, collectors of statistics, defect trackers, and so on. However, facing the substantial complexity of many current software systems, *build systems* have become one of the most important sets of tools. A build system takes care of maintaining a graph of configurations (of varying release status), including all information required to build the actual deliverables whenever needed. Industrial strength build systems tend to apply extensive consistency checks, including automated runs of test suites, on every "check in" of new code.

[39] Where subsystem composition is guided by architecture, those system properties that were successfully considered by the architect are achieved by construction rather than by observing rather randomly emerging composition properties. For example, a security architecture may put reliable trust classifications in place that prevent critical subsystems from relying on arbitrary other subsystems. Otherwise, following this example, the security of an overall system often is as strong as its weakest link.

[40] Other such properties are interface abstraction (hiding all irrelevant detail at interfaces) and encapsulation (hiding internal implementation detail).

[41] The internal modularization of higher-level modules exploits this lack of cohesion. The coarse grain modularity at the top is a concession to human understanding and to industrial organization, where the fine-grain modularity at the bottom is a concession to ease of implementation. The possibility of hierarchical decomposition makes strong cohesion less important than weak coupling.

[42] By atomic, we mean an action cannot be decomposed for other purposes, although it can be customized by parameterization. On the other hand, a protocol is composed from actions. An action does not require an operation in the module invoking that action (although such an operation may follow from the results of the action). A protocol, on the other hand, typically coordinates a sequence of back-and-forth operations in two or more modules, in which case it could not be realized as a single action.

[43]Interfaces are the dual to an architect's global view of system properties. An interface determines the range of possible interactions between two modules interacting through that interface and is thus narrowing the viewpoint to strictly local properties. Architecture balances the dual views of local interaction and global properties by establishing module boundaries and regulating interaction across these boundaries through specified interfaces.

[44] Encapsulation requires support from programming languages and tools.

[45] This terminology arose because the interface between an application and operating system was the first instance of this. Today, the term API is used in more general contexts, such as between two applications.

[46] Sometimes "emergence" is used to denote unexpected or unwelcome properties that arise from composition, especially in large-scale systems where very large numbers of modules are composed. Here we use the term to denote desired as well as unexpected behaviors. An example of emergence in the physical world is the airplane, which is able to fly even though each of its subsystems (wings, engines, wheels, etc.) is not.

[47] Bits cannot be moved on their own. What is actually moved are photons or electrons that encode the values of bits.

[48] Imagine a facsimile machine that calls the answering machine, which answers and stores the representation of the facsimile in its memory. (This is a simplification with respect to a real facsimile

machine, which will attempt to negotiate with the far-end facsimile machine, and failing that will give up.) Someone observing either this (simplified) facsimile machine or the answering machine would conclude that they had both completed their job successfully—they were interoperable—but in fact no image had been conveyed.

[49] A Web browser and a Web server need to interoperate in order to transfer the contents of Web pages from the server to the browser. However, once transferred, the browser can go offline and still present the Web page for viewing, scrolling, printing, etc. There is not much need for any complementarity beyond the basic assignment of the simple roles of page provisioning to the server and page consumption to the browser.

[50] In more complicated scenarios, Web pages contain user-interface elements. The actual user interface is implemented by splitting execution between local processing performed by the browser and remote processing performed by the server. To enable useful user interfaces, browsers and servers need to complement each other in this domain. Browser and server compose to provide capabilities that neither provides individually.

[51] In even more involved scenarios, the Web server can send extension modules to the browser that extends the browser's local processing capabilities. Java applets, ActiveX controls, and browser plug-ins (such as Shockwave) are the prominent examples here. For such downloadable extension modules to work, very tight composition standards are required.

[52] Of course, one common function of software is manipulating and presenting information content. In this instance, it is valued in part for how it finds and manipulates information.

[53] This assertion is supported by numerous instances in which software, supported by the platform on which it executes, directly replaces physical products. Examples include the typewriter, the game board, the abacus, and the telephone.

[54] For example, each individual drawing in a document, and indeed each individual element from which that drawing is composed (like lines and circles and labels), is associated with a software module created specifically to manage that element.

[55] Technically, it is essential to carefully distinguish those modules that a programmer conceived (embodied in source code) from those created dynamically at execution time (embodied as executing native code). The former are called *classes* and the latter *objects*. Each class must capture various configuration options as well as mechanisms to dynamically create other objects. This distinction is also relevant to components, which are described in Section 6.5.2.

[56] For many applications, it is also considered socially mandatory to serve all citizens. For example, it is hard to conceive of two Webs each serving a mutually exclusive set of users.

[57] This is particularly valuable for upgrades, which can be distributed quickly. This can be automated, so that the user need not take conscious action to upgrade his or her programs. Popular examples here are the Web-based update services for Windows and Microsoft Office.

[58] Mobile code involves three challenges beyond simply executing the same code on different machines. One is providing a platform that allows mobile code to access resources such as files and display in the same way on different machines. Another is enforcing a set of (usually configurable) security policies that allow legitimate access to resources without allowing rogue code to take deleterious actions. A third is to protect the mobile code (and the user it serves) from rogue hosting environments. Today, this last point is an open research problem.

[59] This enhances the *scalability* of an application, which is the ability to cost-effectively grow the facilities so as to improve performance parameters in response to growth in user demand.

[60] The data generated by a program that summarizes its past execution and is necessary for its future execution is called its *state*. A mobile agent thus embodies both code and state.

[61] The choice of object code and interpreter is subject to direct network effects. Interpreters (e.g. the JVM) are commonly distributed as part of the operating system. Fortunately, it is possible to include two or more interpreters, although this would complicate or preclude composition on the target platform.

[62] An example is the World-Wide Web Consortium (W3C), which is a forum defining standards for the evolution of the Web.

[63] A reference model is determined as the first step in a standards process. Sometimes the location of open interfaces is defined instead by market dynamics (e.g. the operating system to application).

[64] An obvious example is the hierarchical decomposition of a reference-model module, which is always an implementation choice not directly impacting consistency with the standard.

[65] More specifically, specifying interfaces focuses on interoperability, and specifying module functionality emphasizes complementarity, together yielding composability (see Section 3.3.5).

[66] Examples include the Windows operating system API and the Hayes command set for modems.

[67] Unfortunately, this is not all that far from reality—the number of interfaces used concurrently in the present software (and hardware) world is substantial.

[68] The IETF has always recognized that its standards were evolving. Most IETF standards arise directly from a research activity, and there is a requirement that they be based on working experimental code. One approach used by the IETF and others is to rely initially on a single implementation that offers open-world extension "hooks". Once better understood, a standard may be "lifted" off the initial implementation, enabling a wider variety of interoperable implementations.

[69] Technically, this is called semantic tiering.

[70] This does not take account of other functions that are common with nearly all businesses, like marketing (related to Section 2) and distribution (discussed in Section 6.2.2).

[71] Often infrastructure hardware and software are bundled together as equipment. For example, the individual packet routing is implemented in hardware, but the protocols that configure this routing to achieve end-to-end connectivity are implemented in software. The boundary between hardware and software changes over time. As electronics capabilities outstrip performance requirements, software implementations become more attractive.

[72] While supporting the needs of *all* applications is an idealistic goal of infrastructure, this is rarely achieved in practice. This issue is discussed further in Section 6.

[73] Performance is an issue that must be addressed in both the development and provisioning stages. Developers focus on insuring that a credible range of performance can be achieved through the sizing of facilities (this is called *scalability*), whereas provisioning focuses on minimizing the facilities (and costs) needed to meet the actual end-user requirements.

[74] Some of these functions may be outsourced to the software vendor or third parties.

[75] An example is Enterprise Resource Planning (ERP) applications, which support many generic business functions. ERP vendors provide modules that are both configurable and can be mixed and matched to meet different needs.

[76] This process is more efficient and effective when performed by experienced personnel, creating a role for consulting firms that provide this service.

[77] Mainframes have not disappeared, and continue to be quite viable, particularly as repositories of mission critical information assets.

[78] One difference is the greatly enhanced graphical user interface that can be provided by desktop computers, even in the centralized model. Another is that today the server software focuses to a greater extent on COTS applications, providing greater application diversity and user choice, as compared to the prevalence of internally developed and supported software in the earlier mainframe era.

[79] Such controls may be deemed necessary to prevent undetectable criminal activity and to prevent the export of encryption technology to other nations.

[80] Open source software, discussed later, demonstrates that it is possible to develop software without financial incentives. However, this is undoubtedly possible only for infrastructure software (like operating systems and Web browsers) and applications with broad interest and a very large user community.

[81] "Productive use" sees many different definitions, from frequent use to high duration of use.

[82] In practice, there is a limited time to pass on unauthorized copies of software to others. In the longer-term, object code will almost certainly fail to run on later platforms or maintain its interoperability with complementary software. The continuing maintenance and upgrade is a practical deterrent to illegal

copying and piracy. Another is the common practice to offer substantial saving on upgrades, provided a proof of payment for the original release can be presented.

[83] Source code is sometimes licensed (at a much higher price than object code) in instances where a customer may want or need the right to modify. In this case, the supplier's support and maintenance obligations must be appropriately limited. In other cases, source code may be sold outright.

[84] Sometimes, the source comes with contractual constraints that disallow republication of modified versions or that disallow creation of revenue-generating products based on the source. The most aggressive open source movements remove all such restrictions and merely insist that no modified version can be redistributed without retaining the statements of source that came with the original version.

[85] Scientific principles and mathematical formulas have not been patentable. Software embodies an algorithm (concrete set of steps to accomplish a given purpose), which was deemed equivalent to a mathematical formula. However, the mutability of software and hardware—both of which can implement algorithms—eventually led the courts to succumb to the patentability of software-embodied inventions.

[86] Software and business process patents are controversial. Some argue that the software industry changes much faster than the patent system can accommodate (both the dwell time to issuing and the period of the patent). The main difficulty is the lack of a systematic capturing of the state of the art through five decades of programming, and the lack of history of patents going back to the genesis of the industry.

[87] Open source is an interesting (although limited) counterexample.

[88] The purpose of composition is the emergence of new capabilities at the systems level that were not resident in the modules. The value associated with this emergence forms the basis of the system integration business.

[89] It is rarely so straightforward that existing modules can be integrated without modification. In the course of interoperability testing, modifications to modules are often identified, and source code is sometimes supplied for this purpose. In addition, there is often the need to create custom modules to integrate with acquired modules, or even to aid in the composition of those modules.

[90] An ISP is not to be confused with an Internet service provider, which is both an ISP (providing backbone network access) and an ASP (providing application services like email).

[91] For example, an end user may outsource just infrastructure to a service provider, for example an application hosting service (such as an electronic data processor) and a network provider. Or it may outsource both by subscribing to an application provided by an ASP.

[92] The ASP Industry Consortium (www.aspindustry.org) defines an ASP as a firm that "manages and delivers application capabilities to multiple entities from a data center across a wide area network (WAN)." Implicit in this definition is the assumption that the ASP operates a portion of the infrastructure (the data center), and hence is assuming the role of an ISP as well.

[93] Increasingly, all electronic and electromechanical equipment uses embedded software. Programmable processors are a cost effective and flexible way of controlling mechanisms (e.g. automotive engines and brakes).

[94] For example, where there are complementary server and client partitions as discussed in Section 6.4.3, the server can be upgraded more freely knowing that timely upgrade of clients can follow shortly. A reduction of the TCO as discussed in Section 4.2.3 usually follows as well.

[95] The mobile code option will typically incur a noticeable delay while the code is downloaded, especially on slow connections. Thus, it may be considered marginally inferior to the appliance or ASP models, at least until high speed connections are ubiquitous. The remote execution model, on the other hand, suffers from round-trip network delays, which can inhibit low-latency user interface feedback, such as immediate rotation and redisplay of a manipulated complex graphical object.

[96] Embedded software operates in a very controlled and static environment, and hence is largely absent operational support.

[97] Mobile code may also leverage desktop processing power, reducing cost and improving scalability for the ASP. However, there is a one-time price to be paid in the time required to download the mobile code.

[98] In order to sell new releases, suppliers must offer some incentive like new or enhanced features. Some would assert that this results in "feature bloat", with a negative impact on usability. Other strategies include upgrading complementary products in a way that encourages upgrade.

[99] If a software supplier targets OEMs or service providers as exclusive customers, there is an opportunity to reduce development and support costs because the number of customers is smaller, and because the execution environment is much better controlled.

[100]Depending on the approach taken, pay-per-use may require significant infrastructure. For example, to support irregular uses similar to Web browsing at a fine level of granularity, an effective micro-payment system may be crucial to accommodate very low prices on individual small-scale activities.

[101] This actually is based on software components (see Section 6.5.2). Each component has encapsulated metering logic, and uses a special infrastructure to periodically (say, once a month) contact a billing server. In the absence of authorization by that server, a component stops working. The model is transitive in that a component using another component causes an indirect billing to compensate the owner of the transitively used component. Superdistribution can be viewed as bundling of viral marketing [War00] with distribution and sale.

[102] A high percentage (estimates range from 40% to 60%) of large software developments are failures in the sense that the software is never deployed. Many of these failures occur in end-user internal developments. There are many sources of failure—even for a single project—but common ones are an attempt to track changing requirements or a lack of adequate experience and expertise.

[103] Software suppliers attempt, of course, to make their applications as customizable as possible. Usually this is in the form of the ability to mix and match modules, and a high degree of configurability. However, with the current state of the art, the opportunity for customization is somewhat limited.

[104] Here too, there are alternative business models pursued by different software suppliers. Inktomi targets Internet service providers, providing all customers of the service provider with enhanced information access. Akamai, in contrast, targets information suppliers, offering them a global caching infrastructure that offers all their users enhanced performance.

[105] This places a premium on full and accurate knowledge of the infrastructure API's. Customer choice is enhanced when these API's are open interfaces.

[106] An example of a similar layering in the physical world is the dependence of many companies on a package delivery service, which is in turn dependent on shipping services (train, boat, airplane).

[107] Examples are: the Java bytecode representing a program, a relational table representing structured data for storage, and an XML format representing data for communication.

[108] Examples are: the instructions of a Java virtual machine, the SQL operators for a relational database, and the reliable delivery of a byte stream for the Internet TCP.

[109] An example is directory services, which combines communication and storage.

[110] For example, applications should work the same if the networking technology is Ethernet or wireless. Of course, there will inevitably be performance implications.

[111]This is analogous to standardized shipping containers in the industrial economy, which serve to allow a wide diversity of goods to be shipped without impacting the vessels.

[112] By stovepipe, we mean an infrastructure dedicated to a particular application, with different infrastructure for different applications.

[113] Examples are the failed efforts in the telecommunications industry to deploy video conferencing, videotext, and video-on-demand applications. In contrast, the computer industry has partially followed the layering strategy for some time. For example, the success of the PC is in large part attributable to its ability to freely support new applications.

[114] In many cases, there is a web of relationships (for example the set of suppliers and customers in a particular vertical industry), and bilateral cooperation is insufficient. An additional complication is the constraint imposed by many legacy systems and applications.

[115] This is similar to the layering philosophy in Figure 6. Suppose *N* different representations must interoperate. A straightforward approach would require $N*(N-1)$ conversions, but a common intermediate representation reduces this to $2N$ conversions.

[116] Software is reused by *using* it in multiple contexts, even simultaneously. This is very different from the material world, where reuse carries connotations of recycling and simultaneous uses are generally impossible. The difference between custom software and reusable software is mostly one of likelihood or adequateness. If a particular module has been developed with a single special purpose in mind, and either that purpose is a highly specialized niche or the module is of substantial but target-specific complexity, then that module is highly unlikely to be usable in any other context and is thus not reusable.

[117] However, the total development cost and time for reusable software is considerably greater than for custom software. This is a major practical impediment. A rule of thumb is that a reusable piece of software needs to be used at least three times to break even.

[118] For example, enterprise resource planning (ERP) is a class of application that targets standard business processes in large corporations. Vendors of ERP, such as SAP, Baan, Peoplesoft, and Oracle, use a framework and component methodology to try to provide flexibility.

[119] The closest analogy to a framework in the physical world is called a platform (leading to possible confusion). For example, an automobile platform is a standardized architecture, and associated components and manufacturing processes that can be used as the basis of multiple products.

[120] Infrastructure software is almost always shared among multiple modules building on top of it. Multiple applications share the underlying operating system. Multiple operating systems share the Internet infrastructure. Traditionally, applications are also normally shared—but among users, not other software.

[121] Even an ideal component will depend on some platform for a minimum of the execution model it builds on.

[122] Market forces often intervene to influence the granularity of components, and in particular sometimes encourage course-grain components with considerable functionality bundled in to reduce the burden on component users and to encapsulate implementation details.

[123] A component may "plug into" multiple component frameworks, if that component is relevant to multiple aspects of the system.

[124] This is similar to the argument for layering (Figure 6), common standards (Section 6.4.2), and commercial intermediaries, all of which are in part measures to prevent a similar combinatorial explosion.

[125] Thus far there has been limited success in layering additional infrastructure on the Internet. For example, the Object Management Group was formed to define communications middleware but its standards have enjoyed limited commercial success outside coordinated environments. Simply defining standards is evidently not sufficient.

[126] There are some examples of this. The Web started as an information access application, but is now evolving into an infrastructure supporting numerous other applications. The Java virtual machine and XML were first promulgated as a part of the Web, but are now assuming an independent identity. The database management system (DBMS) is a successful middleware product category that began by duplicating functions in data management applications (although it also encounters less powerful network externalities than communications middleware).

[127] If a server-based application depends on information content suppliers, its users may benefit significantly as the penetration increases and more content is attracted.

[128] Examples in the peer-to-peer category are Napster [Sul99] and Groove Transceiver [Bul00]. As downloadable software, Napster was relatively successful; if an application is sufficiently compelling to users, they will take steps to download over the network.

[129] Under simple assumptions, the asset present value due to increased profits of a locked-in customer in a perfectly competitive market is equal to the switching cost.

[130] They also force the customer to integrate these different products, or hire a systems integrator to assist.

[131] See Section 7.2.5 for further clarification. In a rapidly expanding market, acquiring new customers is as important or more important than retaining existing ones.

[132] Pure competition is an amorphous state of the market in which no seller can alter the price by varying his output and no buyer can alter it by varying his purchases.

[133] An exception is a software component, which may have a significant asset value beyond its immediate context.

[134] Some examples are CBDIForum, ComponentSource, FlashLine, IntellectMarket, ObjectTools, and ComponentPlanet.

[135] For example, office suites offer more convenient or new ways to share information among the word processor, presentation, and spreadsheet components.

[136] In many organizational applications, maintenance is a significant source of revenue to suppliers.

[137] The .NET Framework is an example of a platform that supports side-by-side installation of multiple versions of a component.

[138] For example, bundling an inexpensive and encapsulated computer with Web browsing and email software results in an appliance that is easier to administer and use than the PC. IOpener is a successful example [Net]. The personal digital assistant (PDA) such as the Palm or PocketPC is another that targets personal information management.

[139] This last point is controversial, because information appliances tend to proliferate different user interfaces, compounding the learning and training issues. Furthermore, they introduce a barrier to application composition.

[140] This is only partially true. Especially when appliances are networked their embedded software can be maintained and even upgraded. However, it remains true that the environment tends to be more stable than in networked computing, reducing the tendencies to deteriorate and lessening the impetus to upgrade.

[141] Examples include audio or video equipment, game machines, and sporting equipment. Embedding email and Web browsing capabilities within the mobile phone is another example.

[142] Jini, which is based on Java, and Universal Plug-and-Play, which is based on Internet protocols, are examples of technical approaches to interoperability in this context.

[143] A practical limitation of wireless connections is reduced communication speeds, especially relative to fixed fiber optics.

[144] It may be necessary or appropriate to allow application code to reside within the network infrastructure. Mobile code is a way to achieve this flexibly and dynamically.

[145] An important exception is a *product line architecture* that aims at reusing components across products of the same line. Here, product diversity is the driver, not outsourcing of capabilities to an external component vendor.

[146] An example would be to use a universal remote control to open and close the curtains, or a toaster that disables the smoke detector while operating.