Marketplace Issues in Software Planning and Design

David G. Messerschmitt, *University of California, Berkeley*

Clemens Szyperski, *Microsoft Research*

Software management and project decisions must consider many factors.[1,2] The most recognized return-on-investment drivers include the value to users; the cost of development, maintenance, upgrades, and customer support; and the risks of project failure, budget overrun, and revenue shortfall. This article focuses on other ROI issues related to positioning software in the marketplace, which serves as an intermediary between a software supplier organization and its customers.

Not all marketplace issues (for example, sales channels) strongly affect software project planning and decision-making, but the decisions we discuss here do. Rather than discuss contracted software development for specific customers or open-source software efforts, we'll emphasize the uncertainties and strategic issues that suppliers encounter in selling or licensing software in the general marketplace.[3,4] We categorize these issues by the standard ROI measures of revenue, cost, and risk.

Strategic decisions in planning and designing commercial software often arise from marketplace issues related to the ROI measures of revenue, cost, and risk. These issues should influence ongoing project planning and design decisions and should not be left to business managers alone.

Revenue

Revenue is affected by many factors, including pricing strategies and unit sales. All these factors strongly depend on design decisions affecting customer-perceived value and customer-incurred costs, as well as the status of competition. Factors deserving careful consideration include the customer's costs of switching to competitors, the value of compatibility with complementary products, and any value associated with more widespread adoption.

Companies can increase revenue by charging higher unit-license or subscription fees, by increasing the number of units licensed, or both. Total revenue is proportional to the product of market share and market size, and many revenue maximization strategies increase one at the expense of the other.

Increasing market share requires taking customers away from competitors or attracting a disproportionate share of new sales; the latter is especially effective in rapidly growing markets. Economic theory suggests that, assuming equally effective marketing and sales efforts, customers gravitate toward the products with the largest *consumer surplus*—the difference between value (willingness to pay) and total cost of ownership, including recurring operational and acquisition costs. Software design should therefore focus on both increasing value and reducing the customer's total cost. In all cases, the focus should be on the paying customer (often an organization), not just the software's users. Although usability is obviously a significant source of value, the customer often has a larger agenda. For example, the customer might want to increase productivity, which can reduce the number of users. Moreover, the customer has many costs other than software acquisition—for example, provisioning, deployment (including process changes and training), and administration and management of the software installation. Systematically reducing recurring operational costs and the costs of business process changes and training offers considerable leverage because these costs often dwarf the cost of software acquisition. Reducing customer costs allows increased prices (more revenue) and leads to increased surplus (competitive advantage and more unit sales).

Increasing the overall market size often requires cooperation with competitors,[5] yielding an explicit compact to reduce market share for each competitor while increasing aggregate revenues. Here, we discuss each of these strategies, emphasizing the important issues in designing and developing software.

Increasing value

Increasing value and satisfaction for customers is critical for remaining competitive but often isn't sufficient to take market share away from an established competitor. The reason is that software has high first-copy costs and low replication and distribution costs, creating substantial economies of scale (unit cost decreasing with volume), although some recurring costs such as distribution and customer support do increase with unit sales. Suppliers should avoid creating direct substitutes for other vendors' products, partly because the economies of scale for software strongly favor the supplier with the larger market share: Established suppliers have higher revenue with comparable costs. Obviously, substantial differentiation from competitors in many possible dimensions—features, usability, provisioning, and operations costs—is crucial.[4] Conversely, there are considerable advantages to being first to market with a new product category or a significant enhancement. Software design should therefore focus heavily on useful, customer-valued innovation.

Network effects. In the presence of network effects, the total number of adopters typically increases a software product's value.[4,6] These effects can be *positive* (for example, the value of instant messaging to each user increases with the number of participating users) or *negative* (the value of the Internet at a fixed capacity decreases if it becomes congested due to more users). They can also be *direct* (product instances are directly dependent) or *indirect* (value depends on some complementary commodity such as the number of programmers facile with a given language or the available content in an information access application). Network effects have become more pervasive in post-Internet software because, within a distributed system, the components often have a direct mutual dependence. They can stifle a new product in its infancy or enable its adoption to take off explosively once it reaches a critical mass of adopters. A *virtuous cycle*, in which success feeds upon success by reinforcing network effects, can increase market size—for example, the rapid adoption encountered by the original Napster.

A software development can account for network effects by either expanding the base of adopters, often at the expense of market share or unit prices, or consciously minimizing the dependence of value on market size. Choosing proprietary interfaces for similar or overlapping products might fragment the base of adopters, whereas choosing an open interface can let the product tap into an existing network. For example, with Web services, competitors have mutually benefited from cooperating on a common platform for distributed applications.

A distributed application's architecture can strongly affect the network. For example, attracting many adopters to a peer-to-peer architecture is crucial. A client-server architecture, on the other hand, offers first adopters full

Software design should focus on both increasing value and reducing the customer's total cost.

In software, large first-copy costs make unit cost loosely dependent on unit sales, so costs offer little insight into pricing.

value (with notable exceptions, such as a site that tries to create a collaborative community). For instance, Napster depended on rapid adoption for its value, whereas iTunes offered essentially full value to its first adopter. Capabilities such as software downloading with automatic installation or mobile code help overcome the early-adopter challenges of direct network effects (witness Napster or Skype) and enable the establishment of applications that would otherwise be impossible.

Extracting value through pricing. Pricing is the strategic business process for converting value into the greatest revenue,[4] and it drives many design decisions. Pricing alternatives such as negotiating a selling price directly (typical of outsourcing firms), licensing fees for each major upgrade, or basing ongoing subscription fees on usage measures have direct design implications.[3] For example, selling software capabilities as a service limits deployment platform options, thus simplifying design and maintenance. Each pricing scheme demands a different design philosophy, and some require explicit integral monitoring and billing elements, license servers, and so on.

In software, large first-copy costs make unit cost loosely dependent on unit sales, so costs offer little insight into pricing. Thus, suppliers should try to base price not on costs but on customer surplus, considering customer costs as well as value. This is more practical if a product is strongly differentiated from those of competitors, because direct competition drives unit prices toward unit marginal costs, which are below average costs given the economies of scale. A corollary is that it's advantageous to drive down customer costs for complementary products from other suppliers by choosing cheaper options or by encouraging competitive options. For example, through design and support, an operating-system supplier should encourage both a diversity of applications and competitive options for each application. In this case, choice and lower prices for applications both work to the OS supplier's advantage. The OS supplier can encourage applications through attention to the application suppliers' needs with good documentation and support.

Basing pricing on consumer surplus rather than costs has a fundamental challenge: Willingness to pay is different for different users. To maximize revenue, value pricing implies price discrimination (basing unit price on something other than the marginal unit costs incurred on a customer's behalf). Approaches include segmenting customers into groups and charging different prices (for example, student discounts) or negotiating prices directly (for outsourced development or large site licenses). A common approach with strong design implications is creating different product variants, all available for sale at the same time, and letting customers self-select on the basis of their willingness to pay. To support this, the design must explicitly allow different combinations of feature and performance sets that are not user configurable. For example, the lower-price student edition of some applications doesn't make use of a floating-point unit, thus lowering performance. Of course, designers must trade this flexibility against long-term maintenance and support costs, which grow with the number of variants and perhaps even the number of combinations of configuration options.

Another value-pricing strategy is *bundling*, or packaging different software products or feature sets to sell as a unit. The *dispersion* in value attached to the bundle is often less than that of its constituents because different customers place the highest value on different constituents. So, there can be a higher bundled price relative to its constituents. If the bundler's constituents are composable—the whole is functionally greater than the sum of the parts—the value increases further. An example of this would be the exchange of information and its formatting among the components of an office suite. In anticipation of bundling, the design of individual products—both internal and external—should anticipate opportunities to compose its constituent products.

Reducing customer costs

Reducing customer costs (other than for software acquisition) is generally desirable. In choosing a software vendor, customers will consider many factors besides features and usability. Some factors, such as the cost of hardware maintenance contracts, are largely separable from the software design, but most interact strongly with the design. For example, the design of software targeting an organizational user often presumes internal processes, either limiting market share or causing customers to make costly modifications to business processes. Thus, to match variations in local needs, it's important to focus on processes, process-change costs, and flexibility to meet process differences across the customer group. An example is the modular options provided by enterprise-resource-planning vendors such as SAP.

Another category of costs is switching costs, which customers encounter in moving from one vendor's product to that of a competitor. Switching costs are a form of lock-in that let the original supplier charge a higher price for upgrades, maintenance, and support.[4] An example might include a customer who must replace complementary products and infrastructure or else incur costs in losing or transforming data. (Of course, locking in existing customers has the unintended consequence of discouraging new customers, who will likely notice switching costs consciously incorporated into the design by the supplier.) All else being equal, a design should maximize switching costs for customers moving *to* competitors and minimize them for customers moving *from* competitors. An example is providing translations from (but not toward) the data formats of competitors' products.

Customers are aware of lock-in; they are concerned not only with competitive options but also with being stranded with a supplier who abandons maintenance and upgrade or goes out of business. Presenting a credible roadmap for future product evolution is helpful, especially when the customer perceives considerable switching costs. Such a roadmap should be an integral part of product planning for marketing as well as for internal purposes.

Customers increasingly resist large monolithic applications, which minimize a customer's integration costs and the need for support from the supplier but which also constrain available functionality and incur greater switching costs. The alternative is modular solutions with options for mixing and matching products from different suppliers, or modules from the same supplier (the reverse of bundling). For example, enterprise-resource-planning applications typically provide many configuration options as well as the ability to choose capabilities or interoperate with other vendors' products. A major strategic-planning issue is thus whether or not to embrace or resist customer-configurable modularity, open interfaces, and complementarity.

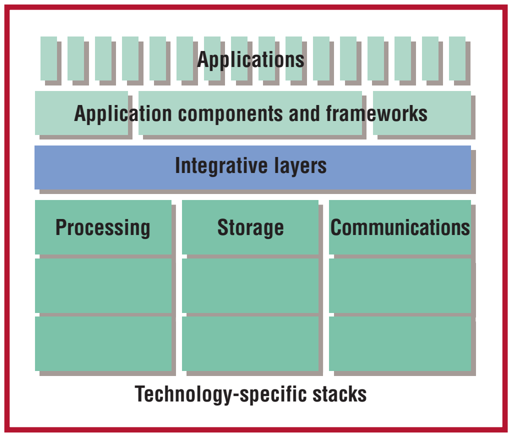
Applications
Application components and frameworks
Integrative layers
Processing
Storage
Communications
Technology-specific stacks

Figure 1. Top-level software architecture.

Increasing vendor cooperation

Encouraging complementary solutions from other vendors increases value, although with little differentiation from competitors who can exploit the same complements. Two ways to encourage complementary solutions with serious business implications are standardization and APIs. A standardized interface can open competition on both sides of an interface, whereas an API encourages complementary extensions while trying to maintain a proprietary position on one side. Choosing whether to offer an API or participate in standardization is a technical choice with serious business ramifications.

Architecture. This phase of project planning determines the boundaries of competition and complementarity.[7,8] This is also the stage when most economic considerations arise.[9] Because the boundaries of firms must align with recognized interfaces, these architectural boundaries must also be consonant with the organization of firms in the software industry. Does industrial organization determine large-grained architecture, or vice versa? Large firms tend to define the architecture, whereas small firms position themselves within an architectural niche. Architectural control can be a competitive advantage, so the way a supplier acquires and retains it is an important strategic issue.

At the coarsest grain, software has a layered architecture, as Figure 1 shows. At the top are various applications, and at the bottom are various technologies, each evolving fairly independently because of homogeneous, integrative middle layers.[3] Moving down the layers, products become less application-specific but more technology-dependent; strategy depends on the

An open interface is usually two-way, creating mutual dependence and supporting competitive options on both sides.

supplier positioning within this stack. Those near the top and bottom can more readily differentiate products from competitors because of the diversity of user needs and the diversity of technology possibilities. Homogeneous middle layers offer higher value (network effects again), so differentiation of features or functionality within them is a challenge. Structurally, no supplier provides a total solution, but composition and integration with other layers are necessary to provide value to customers, and coordination is essential.

Composability is also established architecturally. This requires both interoperability (sharing data and participating in shared protocols) and complementarity (having functionalities that work together). Composition can be explicitly built into the design—for example, in integrating infrastructure layers—or can be opportunistic at deployment (making components capable of assembly but still designing them independently). The latter is far more challenging technically and organizationally but offers considerable value.

APIs. Open APIs invite extension through complementary products or context-specific add-ons by the user. However, they also cede market share in the interest of encouraging innovation, broader choice, and customization. *Openness* refers to documentation and right of use without explicit business arrangements or encumbrance by intellectual-property rights. An API creates an implicit, long-term obligation for customer support. For example, Microsoft's VBA extensions to Office include an effective obligation to maintain and support that capability for the product's life.

Standardization. Open interface standards usually arise from industry standardization or de facto from market success followed by broad industry acceptance. An open interface is usually two-way, creating mutual dependence and supporting competitive options on both sides. A third approach to composition is explicit integration of components that a supplier or intermediary acquires in the marketplace.

Proprietary solutions offer greater opportunity for innovation and differentiation from competitors but can cause customer concern about both lock-in and fewer complementarities. Adherence to standards makes market success more predictable, particularly in ensuring interoperability with complements, and is popular with customers. However, such adherence gives the customer an opportunity to mix and match solutions from different suppliers, thus increasing competitive pressures. Standardization processes have become an integral industry-coordination aspect of software engineering.

Cost

A customer's costs indirectly affect a supplier's revenue through unit sales or feasible pricing, but the supplier's costs directly reduce margins. These costs include first-copy development and testing, distribution, customer support, and recurring development (maintenance and upgrade), all strongly impacted by the software's design. Marketplace factors—such as the chosen distribution platforms and methods, reuse strategies, and make-or-buy decisions—strongly influence these costs as well. There are trade-offs between supplier and customer costs; for example, increasing expenditures on support and maintenance can reduce the customer's internal administrative or user-support costs. The effect on margins depends on how reduced customer costs transform to higher revenue, either through higher unit prices or increased unit sales. There are also opportunities to minimize supplier costs at the organizational (as opposed to project) level through resource sharing, software reuse, and buy-or-make decisions. Three important market-related cost factors are platform and distribution, recurring costs, and reuse and multiple use.

Platform and distribution

Any software distribution defines the scope of potential customers by targeting an infrastructure *platform* (a set of capabilities assumed to be available and static) and complementary software assumed available.[3] Porting and maintaining software for more platforms can increase revenues but can also increase development, maintenance, testing, and support costs. Portability through language and platform choice can mitigate some costs. However, it also demands more of each platform (assuming a built-in virtual machine) and homogenizes the infrastructure, making it more difficult to promulgate new infrastructure capabilities. Removing dependencies on assumed complementary software increases reach, but correctly interfacing this software could also increase development and recurring costs.

An application-as-service business model eliminates the customer's platform dependency. The model reduces the supplier's development costs by reducing or eliminating the number of heterogeneous platforms that must be supported. In addition, the model reduces the customer's costs by shifting user support from software supplier to service provider. However, it also introduces new billing and payment challenges and associated development costs.

Software distribution can occur in various inexpensive ways, including removable media with a physical distribution system, network downloads, and mobile code. These differ in terms of the burden placed on the user or support staff, timeliness, and supplier and customer costs. An interesting trend is to release source code, gaining possible help from users in identifying and fixing defects and making it easier for others, including customers, to develop complementary products.

Recurring costs

Software developments incur major supplier recurring costs of maintenance, upgrade, and customer support as long as the software is in use. Withdrawing support is a difficult step that needs lead time and coupling to alternative options for customers. Therefore, maintenance and support are long-term financial burdens accepted at the time of first sale, and should be included in all cost and risk assessments. Measures such as automated upgrade and defect-reporting mechanisms, aimed at reducing recurring costs at the early requirements and architectural phases, can generate substantial long-term cost savings.

The upgrade phase of a product life cycle begins with an existing code base. One goal is to introduce new features—especially because attracting incremental revenues is a high priority, and the most serious competition to a new version comes from older versions, assuming upgrading is optional. Reusing the existing code base can interfere with innovation or at least make it more costly. Upgrades must achieve acceptable backwards compatibility (for example, accepting old and new file formats), lest switching to competing products become as attractive to customers as adopting an upgrade. Thus, upgrades give rise to some of the most difficult cost-revenue design dilemmas.

Reuse and multiple use

An organizational approach to reducing development costs is *code reuse*[10,11] (using or modifying code written for another project). Reuse is not as commonplace as it seems, because creating easily reused code requires considerable added effort, contradicting the on-time, on-budget incentives of project management. The forking inherent in code modification in the next project also compromises potential maintenance and upgrade savings, and it increases the challenges of closing security vulnerabilities discovered later.

Fortunately, a supplier can minimize coding in other ways. One approach is to expand infrastructure to capture the needs of multiple applications (for example, universal authentication). Another approach is to buy or license modules or components, creating a software supply chain.[12] Components are multiple-use modules that a supplier can configure and assemble into many contexts without inherent modifications, thus enabling

- Shared resources and development costs spread over more uses
- Improved quality through more in-place experience
- Incentives to reduce complexity and usability by meeting the needs of multiple applications

Web services compose dynamically linked components exported as network services, combining the concepts of multiuse components and application-as-service models. Incremental upgrade of individual components allows a graceful evolution of functionality. However, converging on component granularities that are appropriate for various domains, from tiny to large subsystems, is a challenge.

Risk

Risk constitutes unplanned deviations of revenue and cost from objectives. Minimizing uncertainties in revenue (especially downside uncertainty) and costs (especially upside) is important. This is because investors in a software firm demand a higher return to account for uncertainty of earnings; that is, they discount valuation in the face of risk. Risk can be desirable if it comes with appropriate compensating rewards, but companies must avoid or mitigate risks that cause deviation of results

Adherence to standards makes market success more predictable, particularly in ensuring interoperability.

Patents are effective only to the extent that ideas are fundamental and not easily circumvented.

without sufficient increases in margins.

Risk management is necessary at the project, organizational, and industry levels. One inevitable uncertainty is unit sales, making gross margin (revenue minus cost) difficult to plan. An example of a risk incurred deliberately in the interest of increased margin is an investment in innovative features with the goal of increased sales. However, a supplier should scrupulously minimize risks that aren't associated with greater margins, such as ballooning development costs, inadvertent encouragement of competition, security holes, privacy violations, and piracy. Such risks can dampen sales or increase recurring costs. Risks that aren't manageable at the project level or that are deliberately incurred can often be mitigated at an organizational level (for example, through diversification of investments) or at the industry level (for example, through industry cooperation on standardization). Customers incur risk as well, and software design should minimize uncertainties in either customer value or costs unless compensatory rewards accrue to that customer.

At the project level

At the project level, risk arises from unanticipated development costs (for example, backtracking and reworking) and uncertainties in revenues (for example, lower-than-expected unit sales).

Real options. Without code reuse, the irreversible nature of most investments in software development increases risk. Iterative software processes, such as the spiral model,[13] are fundamentally risk management techniques because they stage investments, reevaluating their efficacy and dynamically adjusting tactics. *Net present value* analysis, a simplistic methodology for estimating ROI, accounts for uncertainties by calculating average return. *Real options* is a technical methodology from financial economics that provides a theoretical basis for staged irreversible investments,[14,15] improving on NPV by explicitly considering uncertainties at each stage and allowing deferral of staged investment decisions. The real-options methodology defers investment decisions as much as possible, systematically accumulating information to improve the quality of those decisions, and quantifies the resulting benefits.

Competition. Unanticipated competition is a major market risk. Because of the large, fixed first-copy costs in software development, a supplier with a larger market share has an inherent advantage—even with higher costs. Therefore, it's best to avoid direct competition with existing products unless clear advantages in value, as opposed to cost, are evident.[4] The cost advantages of a large market share combined with customer lock-in strategies benefit suppliers that complete development and establish market share earlier. Even in the absence of direct competition, developments should carefully weigh both technical and market windows of opportunity, avoiding being too early or too late.

Exogenous risk factors. It's possible to anticipate and effectively counter many exogenous risk factors that are beyond a supplier's control. Examples are security vulnerabilities and privacy invasions.[16] Cracker attacks can increase maintenance costs for suppliers and operational costs for customers, and customers' security and privacy concerns can stifle demand.

Another risk is that other suppliers could appropriate ideas, steal code, or claim infringement on their own inventions. Appropriate use of intellectual-property rights can minimize these risks.[17] For an invention (roughly a novel and useful idea) incorporated into a software product, the basic options are to make it public (in free or copyrighted form), maintain it as a trade secret, or patent it. Trade secret laws can serve to punish malfeasance through theft of technology or through employees changing jobs and carrying proprietary information to competitors. Strong measures to avoid inadvertent public disclosure of secrets can strengthen these rights.

Trade secrets and copyrights cannot prevent others from developing the same or similar ideas independently. In software, exploiting an idea often means revealing at least the possibilities, and reverse engineering is generally legal. Patents can prevent appropriation of inventions by other suppliers, even should they develop the idea independently or reverse-engineer it, but patents also require prior public disclosure of the invention. Patents are effective only to the extent that ideas are fundamental and not easily circumvented. A portfolio of patents can also defensively discourage or thwart infringement claims by others by enabling counterclaims of infringement.

Further Reading

Pricing of Complementary Goods and Network Effects, Nicholas Economides and V. Brian Viard, tech. report 1812, Graduate School of Business, Stanford Univ., 2003; http://gobi.stanford.edu/researchpapers/detail1.asp?Document_ID=1951.

Economides and Viard construct an economic model to explain the observed discrepancy between pricing strategies for Microsoft's Windows and Office products, in which Office sells at a significantly higher price than Windows despite comparable market shares. Starting from the categorization of products into base and complementary products, the model explains how the interplay of base and complement create positive feedback effects. The model also provides insight into the differences in incentives, depending on whether complementary products are owned by the same or different suppliers.

"Software Economics: A Roadmap," Barry W. Boehm and Kevin J. Sullivan, *Proc. Conf. The Future of Software Engineering*, ACM Press, 2000; http://portal.acm.org/citation.cfm?id=336584&coll=Portal&dl=ACM&CFID=19273781&CFTOKEN=53125957.

Organized as an adjunct event to the 2000 edition of the International Conference on Software Engineering, The Future of Software Engineering gathered and presented topical roadmaps on many subdisciplines of software engineering. In that framework, Boehm and Sullivan helped scope research directions and open issues in software economics. As a starting point, they criticized the traditional overemphasis on software's structural and correctness properties while underestimating the importance of capture and management of requirements and risks. Traditionally, software project management focuses on costs but not enough on realized values, which is measurable only based on how well that project identifies and meets requirements. Looking forward, the authors suggest moving to a value-oriented discipline using the real-options methodology.

Software Ecosystem: Understanding an Indispensable Technology and Industry, David G. Messerschmitt and Clemens Szyperski, MIT Press, 2003.

The material covered in this book is closely related to our article. Of course, the book goes deeper and provides many pointers to the literature.

Projects should therefore systematically identify and examine ideas for novelty and usefulness, identifying those that reflect prior art or are owned by others, and act accordingly.

Piracy (massive illicit replication of software distributions) can be a substantial risk, particularly for consumer software. It helps that software gradually loses its potency over time—this should be consciously maximized in the maintenance and upgrade phases. Technical choices can also have a major impact. For example, software can be self-aware and can contact license servers as it executes, and network upgrades can compare machine signatures against registration records. The older approach of copy protection has proven self-defeating—mainly owing to invasiveness, ineffectiveness, and customer resistance.[18]

At the organizational level

Organizations have risk management tools that are unavailable at the project level. These include diversification and reuse across projects (mitigating the irreversibility of investments). For the former, a portfolio of products achieves more consistent margins, provided the constituent risks associated with that portfolio aren't strongly correlated.[19] A typical organization uses shared resources in managing multiple projects at various phases of completion and faces trade-offs between organizational and project efficiency. The opportunity to allocate available resources among available investment opportunities is another important organizational risk management issue.

At the industry level

Industry cooperation, provided it doesn't violate antitrust laws, is feasible through industry standardization or coordination of complementary investments. Such cooperation can reduce risk by focusing suppliers on composable and complementary solutions, and by reducing customer confusion and resistance.

One major industry risk is the chicken-and-egg conundrum of applications and infrastructure.[3] Upgrading and widely deploying infrastructure capabilities offers little value without applications requiring those capabilities. Applications dependent on advanced infrastructure must cope with a smaller installed base and greater costs borne by customers upgrading their infrastructure. Coping strategies include infrastructure and applications suppliers coordinating investments or an infrastructure supplier seeding the market for applications. A recent example is the wide industry cooperation in the standardization and deployment of Web services.

Because revenue, cost, and risk interact and have trade-offs, high-quality decision-making depends on the quantitative modeling of costs, benefits, and uncertainties that considers these trade-offs. Better communication between software engineers, project managers, and business managers, through greater familiarity with one another's domains and a common vocabulary, will enhance overall project success.

References

1. B. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.
2. R. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 2000.
3. D. Messerschmitt and C. Szyperski, *Software Ecosystem: Understanding an Indispensable Technology and Industry*, MIT Press, 2003.
4. C. Shapiro and H. Varian, *Information Rules: A Strategic Guide to the Network Economy*, Harvard Business Press, 1998.
5. A. Brandenburger and B. Nalebuff, *Co-Opetition: A Revolution Mindset That Combines Competition and Cooperation*, Doubleday, 1997.
6. O. Shy, *The Economics of Network Industries*, Cambridge Univ. Press, 2001.
7. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.
8. P. Clements, R. Kazman, and K. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2002.
9. R.J. Kazman, J. Asundi, and M. Klein, "Quantifying the Costs and Benefits of Architectural Decisions," *Proc. 23rd Int'l Conf. Software Eng.* (ICSE 01), IEEE CS Press, 2001, pp. 297–306.
10. H. Mili et al., *Reuse-Based Software Engineering: Techniques, Organizations, and Controls*, John Wiley & Sons, 2001.
11. I. Jacobson, M. Griss, and P. Jönsson, *Software Reuse: Architecture, Process and Organization for Business Success*, Addison-Wesley, 1997.
12. C. Szyperski, *Component Software*, 2nd ed., Addison-Wesley, 2002.
13. B. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, May 1988, pp. 61–72.
14. K. Sullivan, "Software Design: The Options Approach," *2nd Int'l Software Architecture Workshop, Joint Proc. SIGSOFT 96 Workshop*, ACM Press, 1996, pp. 15–18.
15. L. Trigeorgis, *Real Options: Managerial Flexibility and Strategy in Resource Allocation*, MIT Press, 1996.
16. R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, John Wiley & Sons, 2001.
17. K. Rivette and D. Kline, *Rembrandts in the Attic: Unlocking the Hidden Value of Patents*, Harvard Business Press, 1999.
18. D. Wallach, "Copy Protection Technology Is Doomed," *Computer*, Oct. 2001, pp. 48–49.
19. E. Elton and M. Gruber, *Modern Portfolio Theory and Investment Analysis*, 6th ed., John Wiley & Sons, 2003.

About the Authors

David G. Messerschmitt is the Roger A. Strauch Professor of Electrical Engineering and Computer Sciences at the University of California, Berkeley. His research interests include the impact of business and economics on technology. Messerschmitt received his PhD in computer, information, and control engineering from the University of Michigan. He is a member of the US National Academy of Engineering. Contact him at Dept. of Electrical Engineering and Computer Sciences, 231 Cory Hall, Univ. of California, Berkeley, CA 94720-1770; messer@eecs.berkeley.edu.

Clemens Szyperski, who is affiliated with Microsoft Research, is a software architect at Microsoft and an adjunct professor with the School of Computing Science, Queensland University of Technology, Australia. His research interests center on component software. Szyperski received his PhD in computer science from the Swiss Federal Institute of Technology, Zurich. He is a member of the ACM. Contact him at Microsoft Research, One Microsoft Way, Redmond, WA 98052-8300; cszypers@microsoft.com.