Integrating Value and Utility Concepts into a Value Decomposition Model for Value-Based Software Engineering

Mikko Rönkkö[1], Christian Frühwirth[1], and Stefan Biffl[2]

[1] Helsinki University of Technology, Software Business Laboratory
Otaniementie 17, Espoo, Finland
[2] Vienna University of Technology, Institute of Software Technology,
Vienna, A-1040, Austria
{mikko.ronkko,christian.fruehwirth}@tkk.fi,
Stefan.Biffl@tuwien.ac.at

Abstract. Value-based software engineering (VBSE) is an emerging stream of research that addresses the value considerations of software and extends the traditional scope of software engineering from technical issues to business-relevant decision problems. While the concept of value in VBSE relies on the well-established economic value concept, the exact definition for this key concept within VBSE domain is still not well defined or agreed upon. We argue the discourse on value can significantly benefit from drawing from research in management, particularly software business. In this paper, we present three aspects of software: as a technology, as a design, and as an artifact. Furthermore, we divide the value concept into three components that are relevant for software product development companies and their customers: intrinsic value, externalities and option value. Finally, we propose a value decomposition matrix based on technology views and value components.

Keywords: Value-based software engineering, stakeholder value, software business.

1 Introduction

Researchers focusing on value-based software engineering (VBSE) have suggested that the economic and value perspectives should be integrated into the software engineering processes that until now have had a very technical focus. According to Biffl and his colleagues [1] and Huang and Boehm [2], software engineering is currently performed in a value-neutral setting, where the basis of methods and tools is on supporting development of technology, not on creating business value. This value-neutral approach makes it hard to create products that are valuable to people and make it difficult to make financially responsible decisions.

Based on the work of the Economics-Driven Software Engineering Research (EDSER) community, a VBSE research agenda has emerged aiming to integrate value considerations in all aspects of software engineering, and calling forth the development

© Springer-Verlag Berlin Heidelberg 2009

of tools and methods to support the business side of software engineering [1]. Due to the novel nature of this idea, empirical evidence – currently being called for in software engineering [3] – supporting the feasibility of realizing value-based software engineering is limited.

Some of the most central work in the area of VBSE includes the initial theory of VBSE as presented by Jain and Boehm [4] and calculation methods estimating return on investment (ROI) of software development pioneered by for example Erdogmus, Favaro and Halling [5]. This work and the research in VBSE in general use several concepts and techniques from economics and accounting and apply them to the context of software engineering. While the inclusion of general business and management theories to the research of business aspects in software engineering can be considered a fundamental aspect of VBSE, we argue that the researchers in the area could significantly benefit from adapting more of the findings of the so-called software business research into their work.

Software business, as the authors of this paper define it, is a management research area, which focuses on software firms and develops knowledge to understand how and why these firms succeed. When defined this way, software business and VBSE share the phenomenon of interest but differ in the research paradigm. While researchers involved in VBSE use the engineering paradigm and develop tools and methods to help software firms succeed, researchers operating in software business area examine how and why firms succeed using the social sciences paradigm. Simply put, the models of VBSE are mostly prescriptive, while the software business research considers explanatory models as a central goal. We argue that these viewpoints are different sides of the same coin, and hence there is a great potential for cooperation and knowledge sharing.

The rest of the paper is structured as follows. First, we present some of the key theories used in the emerging software business research, particularly those related to value. Second, we review three different ways to conceptualize software based on the current paradigm employed by software business researchers. Throughout these two sections, we use software product development or market driven software development as the context. Finally, we will integrate the perspectives of value and the perspectives of software into a value decomposition matrix. The main contributions of this paper include linking VBSE and software business research as well as providing a conceptual tool to aid in different value considerations.

2 Concept of Value in Software Business Research

The concept of value is central to VBSE. Indeed, the main goal of this research movement is to assign a measure of value on decision making in the software process. This is seen as a complement to previous software engineering research that has mainly focused on technical aspects such as quality, cost, and development time. The concept of value is not strictly defined, but can be evaluated for example by a technique called Stakeholder Value Proposition Elicitation and Analysis [6]. From the perspective of economics, this resembles analyzing the utility function [see e.g. 7] of each stakeholder. After this, win-win technique, can be applied to negotiate the requirements.

Research by Briggs and Grünbacher [8] and Oza, Biffl, Fruehwirth, and Selioukova [9] have demonstrated the successful use of this approach in the elicitation of stakeholder values.

Another set of techniques focuses on valuating different features, requirements, or decisions on a single-dimensional measure (most commonly money) using mathematical formulas adapted mainly from accounting and finance[5]. However, the vagueness of the concept of value seems to be a central problem. If the researchers cannot agree on a common definition of value, we run the risk of producing incommensurable research, which seriously inhibits the progress of the field. We attempt to clarify the concept of value by anchoring it to the theory base used in software business.

The economic concept of value is most commonly defined as the amount of money that a unit of goods or services is traded for. Utility, on the other hand, is all the good and desirable that is created by consuming a product or a service. Hence the concept of value in VBSE is closer to economic utility than economic value. To avoid confusion with the terminology, we use the term "value" for value in VBSE context, and "economic value" when discussing the economic concept. The problem with utility, and value, is that good and desirable are highly subjective and idiosyncratic issues. Economics has solved the problem of diversity in utility between consumers by developing a multi-attribute utility theory [10] and using statistical distribution functions as utility functions. However, the abstract and generic nature of these theories limits their applicability to VBSE, as long as no relevant agreed on dimensions and measures exist for value components. In this paper we omit the philosophical of definition of value and assume that value exists, and we can use any definition that suits our needs. Hence, we rather ambiguously define "value is the degree of desirability". Agreeing that this definition sheds little normative light on the decision-making processes, we will now take a closer look what value means in different contexts. The discussion is structured around two central players in the software markets: utility-seeking customers and profit-seeking firms.

2.1 Values of Utility-Seeking Customers

From the customer perspective the value of software comes from its use, the utility it can create. While this is a seemingly trivial argument, it embeds much complexity: First, the utility is not only dependent on the intrinsic properties of the software, but also the skills of the user and several factors that are external to both the user and the software. Second, as discussed earlier, each customer values the software differently depending on for example her unique set of capabilities, and her own desires for different types of utility. Recent research by Oza and his colleagues [9] illustrates this value diversity in dynamic settings of software process improvement initiatives.

In a static setting where future is not considered, the value of the software comes from three different sources: intrinsic value, complementary value [11] and direct network externalities [12,13]. The intrinsic value is embedded in the software as functionality and attributes such as security and usability. This part of the value seems to have the closest match with the current concept of value in VBSE. A complement can be defined as a product or a service, which increases the value of another product or service [14] and here complementary value refers to value, which is created by combining a piece of software with another good or service. For example, a word processor is much more valuable when bundled in an office suite due to the possibility to embed objects created with other applications. Last, if the software can be used in communication, it is subject to network externalities – its value is dependent on the amount of other users of the software that are relevant to the focal user. For example, if two colleagues use compatible word processors, they can share files and collaborate benefiting both from the compatibility. A somewhat idealistic view of this phenomenon is known as Metcalfe's law "the value of a network is the number of users squared" [15]. Empirical evidence suggests that the value of network externalities can be in par with product features when a product's economic value is evaluated[11,16,12]. In other words, compatibility with other pieces of software can be as important as the features and quality of the software.

The problem with the above-presented decomposition of value is that it does not take into account the bounded rationality of people. Especially in the context of complex issues, people cannot base their decisions fully on facts. Hence, the purchase decisions are not based on real value, but perceived value [see e.g., 17]. We will first discuss the issue of where the estimate of real value comes from and then present some of the factors that may create bias between the real and the perceived value. In economics, a good whose value cannot be estimated without using the good is called "an experience good" [18]. While software is far from typical experience goods, like music, Messerschmitt and Szyperski [19] argue that software should be considered to be an experience good. This implies that the estimate for the value comes from using the good, referring to other users or reviews, or simply through advertisements. In the context of software, especially influential seems to be the experience with the prior generation or release of the product. However, even with perfect information gained through experience, the perceived value rarely equals the real value. One reason for this is that value contains also purely psychological parts. Often the market share correlates with the perceived value causing bandwagon effect [20], where the current user base drives adoption without any mechanism that would generate externalities. While the psychological part of the value has traditionally been considered as being solely in the domain of marketing, some recent work suggests that it should be taken into account also in the product development phase [21]. The significance of these psychological effects can be so strong as to enable firms with inferior products to capture the markets if they gain control of the bandwagon [13]. Product launch timing is an influential factor in creating these effects and hence at least release planning is affected by this market effect [22].

The last problem with estimating the value of a piece of software from the customer point of view is that software is an investment in a durable good and hence the expected future value matters. More concretely, the customer is interested on availability of complements in the future (including for example updates), and expected size of the user base. These both are issues, which can significantly affect which products are chosen and which firms' offerings prevail in the markets.

2.2 Values of Profit-Seeking Firms

Next we will discuss the concept of value from the perspective of a software firm. The objective of the firm is simple: to maximize the cumulative long term profits. However, this simple and uniform concept of value does not help much when trying to estimate the value of software development decisions. The reason is that the profits of the firm are realized in market transactions, and the evolution of the markets is an external facto that is largely outside the control of the firm. This is particularly true for the turbulent software markets, where standards, technologies, and even companies change rapidly. Due to this, attempts to generate systematic heuristics to optimize against the unknown future have not yet matched the use of managerial intuition in decision-making [23].

We will divide the further discussion of the value from the perspective of the software firm into two themes: market mechanisms and path dependency. In economics, market is a place where buyers and sellers exchange goods and services. If a buyer considered something as being more valuable than the seller, a transaction occurs. The purpose of the market is to create and divide surplus – the utility of the good for the buyer measured in money minus the cost of creating the good by the seller measured in money. Since the utility of different buyers varies, the seller usually prices the good in such a way that only a certain amount of users want to trade with the price. When a competitor with a similar product arrives, the optimal price that the seller should charge decreases. If the goods are sufficiently similar and there is sufficiently large number of sellers, the basic economic models predict that prices will fall to a level that equals the cost of production and sales by the sellers. If this so-called perfect competition situation occurs, no firm will create profit. To counter the effects of competition firms often deliberately create products that cause lock-in by means of creating extra costs when switching to other vendors or use advertising to make their product seem more advantageous than it actually is [24]. With these tactics, the firm is decreasing the surplus (value minus cost) that goes to the customer to create more profits. The importance of lock-in is that it enables software firms to extract more value from their products than would be possible if consumers could switch to competing products freely, thus explaining the voracious strive for market share in growing markets [25]. The phenomenon of lock-in and existence of network externalities create a challenge for evaluating the value of the software: Often several incompatible standards compete, and the outcome of this battle for dominance cannot be evaluated accurately ex ante [26]. The dilemma of a firm is that while it maximizes utility by being compatible with the dominant network, it can often capture more economic value by excluding competitors from the network by being incompatible with competing solutions [13]. The dilemma of compatibility and limiting the choice of the customer is something rather opposite to the win-win principle [27] used in VBSE. Another problem is that when technology is first developed and then sold at the markets, the value for the technology cannot be accurately defined at the time when the most value affecting development decisions are made since we cannot accurately predict how the market develops in the future [28].

Another issue with firms is that they have technological path dependency. That is, their future technological options are a function of the technology that they currently have in terms of not only technology assets but also knowledge. This means that sometimes firms need to optimize for longer term rather than following the most value-efficient approach for the current customers. If a firm fails to see this, it might end up in technological obsolescence or technological lockout [29].

To summarize the discussion in this section and the previous, we conclude that there probably cannot be a single unidimensional and measurable construct for value, but how value is seen depends on the context. However, we argue that just abstracting the value to a single figure can sometime be too simple solution since three different dimensions of value exits: intrinsic value of the software, externalities through compatibility and complements, and option value by enabling future development paths. Next we will look at the concept of value from a rather different perspective, that of software as a modular technology.

3 Three Perspectives on Software as Technology

After the initial discussion of value, we will now take an orthogonal view on the issue. If we are to understand what value means for software from the perspective of various management disciplines, we need to also understand how these disciplines conceptualize software. It is easy to define software as a technology without further considerations on the general nature of the term. To understand how software is presented in management research, we adopt a definition for the concept of technology by Schilling [30]

> Technology refers to any manner of systematically applying knowledge or science to a practical application … Technology in this context is generally understood to include information technology as well as technology embodied in products, production processes, and design processes.

Since the process of creating and the process of executing are systematic, and there is a practical application for software, we can indeed conclude that software fits well to this definition. The adopted definition links technology intimately but not exclusively to artifacts, that is, technology is both the artifacts that extend our capabilities and the skill to produce and efficiently use them. This definition is much more strict, than defining technology as knowledge that is intended for "use". If defined this broadly, technology would encompass virtually all useful routines and capabilities developed through organizational evolution.

In addition to artifacts and knowledge, technology can be considered from a third perspective: as a design. Design is a "blue print", a type of artifact that acts as a template for producing more artifacts. While not strictly correct, we distinguish between software design and software artifact by defining that software design is technology-in-development and the software artifact is technology-in-use, or technology which is embedded inside a medium and is ready to be executed or traded. We present each of these views in more detail and build link to VBSE.

3.1 Software Artifacts

Most notable property of software artifacts is that they are information. More precisely, software artifacts are a sequence of instructions that is codified in a form that can be interpreted and executed by computer hardware. Information artifacts have several distinct properties: First, information contains always two parts, message and the language, which it is codified with [31]. With software this naturally implies that the codification needs to be compatible with the hardware. However, in contrast to many other information goods, this codification is not readily comprehensible by people, and in the case of interpreted programming languages where the software is distributed in source code format and interpreted to machine language when executed, it still requires considerable effort to comprehend the code[24]. In this way, software does not suffer from the property shared by many other information goods, that is, software can be appropriated even if it has been once disclosed [32]. Hence, software should be considered as an experience good [19,18], but the implications of revelation are much less serious than with other more typical information goods.

Like any other information good, software does not wear out when used. However, it shares a characteristic with knowledge: Knowledge does not wear out, but competition can drive down the price even though the utility has not declined. The value can also diminish through obsoletion[33]. That is, the utility of the information does not decrease, but the market value is decreased through emergence of new and more advanced competing artifacts, or the environment where it is used changes so that the artifact is no longer useful for the purpose it was intended for. The speed of obsoletion can range from rapid to nearly inexistent. For example software that is run on the mainframes of financial institutions can be even several decades old, while anti-virus software needs to be updated several times a day to keep it on an adequate level of capability to block emerging and constantly developing threats. The value implication of this insight is that normal discounting methods that are used when evaluating economic value over time are not sufficient when considering value of software, which will be developed in the future, since the face value of the artifact does not stay constant over the time.

Software artifacts consist of two types of data: instructions for computer hardware and embedded information. The latter includes all text, images, sounds as well as information that is passed to external devices as forms of instructions [19]. The instruction part of the software artifact is what makes software behave like virtual machines that do things [34]. Software goods that consist mainly of instructions can be considered as tools that help people to get jobs done. Usually, when technology enables us to get things done, there emerges a dominant design [35], and hence there is in the longer run little variance in preferences – or the desire for utility - for software that is low in the information content. If there is no service component linked to the software, the offering of one firm scales easily and hence can result in capturing a monopolistic market share.

In contrast, when the embedded information content of the software is high, or the purpose of the software is to present information interactively, the preferences of the consumers behave very differently [see e.g., 36]. This is due to the fact that information and instruction content are valued differently: While the interactive part is valued for what it does, information is valued for what it teaches us or how it influences us [19]. Generally, there is a large variance in preferences for information, for both entertainment and education purposes. Moreover, these types of products suffer somewhat similar issues than information goods, once the users learn the information, the utility of the software artifact decreases. Prime examples of this kind of software artifacts are computer games. Indeed, computer games are no longer programmed, they are designed since the storyline, graphics, and environment of the game grow in importance related to technical aspects of the program [19]. Once a game is released, it might sell for only less than a year after its initial release. Moreover, once a person has completed the game once, his interest in the program is decreased since there is no element of novelty anymore in the information content.

3.2 Software Designs

As a system, technology is a collection of subsystems that are bound together with architecture, and each subsystem can be a system of other subsystems. The key insight from general systems theory is that a system cannot be comprehended as only through its parts, but needs to be considered as a whole.

According to Schilling [30]

> Modularity is a general systems concept: is a continuum describing the degree to which a system's components can be separated and recombined, and it refers both to the tightness of coupling between components and the degree to which "rules" of the system architecture enable (or prohibit) the mixing and matching of components.

The level of modularity in software artifacts varies significantly, and it is not necessarily tied to modularity of the technology, which was used to generate the artifact. That is, a modular technology can result in highly integrated tightly coupled artifact systems. While this seems initially counterintuitive, it becomes clear after one considers the process of compiling software, where several source files are compiled and linked to become one binary executable. In this process the modularity of the technology is decreased and the modules loose their autonomy: it is no longer possible to easily exchange the compiled modules and in order for the system to work as designed, each module needs to work.

However, one software artifact can consist of several (executable and non-executable) files. In this case the system retains part of the modularity of the technology. For such modular product to be realized, several interfaces are required to define the architecture of the system [37,38]. In software, this modular design has several advantages: modular system can be upgraded or modified by exchanging modules to enhanced versions, and documented modular interfaces enable user driven innovation [39,40]. Modern computer games where users can create new scenarios or modifications are a prime example of the latter. Moreover, modularity enables the emergence of complements, which can be a significant source of value for a software product [41,42].

Modularity is a powerful concept, since modular designs include what Baldwin and Clark [38] call "option value". In their work combining the research streams of real options and complexity theory they identify six modular operations: splitting, substituting, augmenting, excluding, inverting, and porting. After developing theoretical measures of value for performing each of the operation, they present history of the computing industry as an example of how modularity works. The problem of modular design is, that while modularity enables more efficiently constructing a product family, it can lead to loosing the control of the design, that is, the parts of the design provided by the original vendor are no longer the value critical elements.

The power of modularity of design is that much of the complexity can be hidden under layers of abstraction. Modularity, measured often as coupling and cohesion in software engineering, has much benefits, including more comprehensible design and as a consequence result in better developer performance [43,44], can boost the innovation rate at each module, and enable better system reconfigurability [38]. However, this comes with a cost: First, even software with well defined architecture and internal interfaces tend to degrade over time. That is, incremental changes break the architecture and make the modules more tightly coupled if efforts are not spent to prevent this. This is a general property of technology and other complex systems and in software engineering it is known as Lehman's law [45]. When a complex system becomes more integrated, it looses its adaptability [46]. Moreover, the links become more numerous and less general, even to an extent that the abstracting effect of the modular system is lost. There is little use in modularity, if the software designer needs to be concerned with the internal structures of modules.

Clearly, not only the requirements, but also the architecture of the software needs to be value-based, if the long run value of the design is to be optimized. Unfortunately, this is not often the case when firms follow the client or market requirements to stay with the competition, especially when developing products on internet time [47].

3.3 Software Knowledge

The final aspect that we take on software as a technology is that of technological knowledge and competence. Currently, knowledge and technological capability are increasingly in the core of creating competitive advantage for companies [21,48,33] in high tech industries, like semiconductors, biotechnology, electronics, and software, where the development costs of new products can form a significant part of the cost structure of the entire company.

Defining the knowledge part of software is not straightforward unless one knows a bit of psychological aspects of programming. Hence, we start by briefly introducing a psychological view of how software is created. When a software engineer starts to write software that conforms to the previously designed requirements, he goes through a series of tasks. First, the problem is analyzed and formalized so that it can be solved with a computer, after which architecture and components of the solution are designed. This designing follows a cognitive problem solving process, where the software engineer combines external and codified knowledge to his own tacit knowledge creating a mental model of the solution [49-51]. After the model of the solution has been created, it is codified into a message using a programming language [31]. The result of the process of programming is a stream of textual information that resides on a computer or a similar platform. In this sense, the software code is only a projection of the solution developed by the programmer. Several finer aspects, especially why something is done like it is, remain tacit. In essence the codified form and the tacit form of software are intimately linked, and in this way software is tied to the people or organization that developed the software.

Clearly this knowledge is valuable and hence knowledge creation should be included in the value considerations, for example through integrating VBSE and experience factory [52], which is a general knowledge management framework for software engineering organizations. The value of knowledge comes from the fact that ability to learn is a function of what is already known and hence software firms who are on the edge of technology development often invest in projects for the main reason of learning. The downside for knowledge creation is, that it can lead to islands of specialization, where only one person or a small group holds a piece of tacit knowledge that is critical to the software development organization. If this happens it gives these employees an edge in the considerations of how the created value should be distributed among the stakeholders, thus enabling a potentially negative impact on the organizational knowledge distribution.

4 Synthesizing the Two Perspectives into a Value Decomposition Matrix

In the previous two sections we presented two views on value. First, we addressed the issue through three value components: intrinsic value, externalities, and option value. Second, we discussed three different views on software: as artifact, as design and as knowledge. Based on this discussion, we propose a value decomposition matrix to aid in considering the different aspects of value. The matrix is shown below in Table 1. Each cell in the cross-section of a view on software (rows) and value sources (columns) contains an illustrative question to aid in utilizing the nine different combinations in value considerations. The current limitation of the matrix is, that it mostly focuses on the view of the value to the customers and the organization, hence largely disregarding the value considerations that are relevant to employees. More work will be needed here in the future to integrate this third stakeholder group into the value decomposition matrix. Moreover, the framework is focused on market-driven development that takes place in software product firms.

Table 1. Value decomposition matrix

	Intrinsic value	Externalities	Option value
Software artifact	What is the direct value of this decision to the users of the software?	What is the indirect value of this decision to the users of the software through enabling connectivity to other users or software components?	What future software acquisition or enhancement options does this development decision provide for the users?
Software design	What is the direct value of this decision to our ability to create software artifacts?[1]	What is the value of this development decision on our ability to create connectivity and compatibility to our software artifacts?	What is the value of this development decision in terms of modular options?
Software knowledge	What do we learn directly by making this decision?	What do other parties that provide value for the users of our software learn if we take this decision?	What kinds of future learning options does this decision enable us to pursue?

[1] Consider that software design can be used to create several different artefacts (e.g. a product line).

Finally, we propose 5 potential avenues of future research in VBSE:

1. External value sources, like complements and network externalities need to be taken into account in value considerations.
2. Modularity, in terms of modular options and as an enabler for maintenance is a significant source for long-term value.
3. Market mechanisms have been the most successful institution in dividing utility in society and they provide a potential avenue for further research in VBSE.
4. Most firms do not create win-win, but win-loose less (firm-customer) situations, if they achieve lock-in. Hence, win-win does not necessarily create the most optimal solution for the stakeholder that has the most power in decision making.
5. Experience factory or some other knowledge management concept should be integrated in VBSE.

References

1. Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., Grünbacher, P.: Value-Based Software Engineering. Springer, Heidelberg (2005)
2. Huang, L., Boehm, B.: How Much Software Quality Investment Is Enough: A Value-Based Approach. IEEE Software 23, 88–95 (2006)
3. Kitchenham, B., Pfleeger, S., Pickard, L., Jones, P., Hoaglin, D., El Emam, K., Rosenberg, J.: Preliminary guidelines for empirical research in software engineering. IEEE Transactions on Software Engineering 28, 721–734 (2002)
4. Jain, A., Boehm, B.: Developing a theory of value-based software engineering. In: Proceedings of the seventh international workshop on Economics-driven software engineering research, pp. 1–5. ACM Press, St. Louis (2005)
5. Erdogmus, H., Favaro, J., Halling, M.: Valuation of Software Initiatives Under Uncertainty: Concepts, Issues, and Techniques. In: Biffl, S., Aurum, A., Boehm, B.W., Erdogmus, H., Grünbacher, P. (eds.) Value-Based Software Engineering, Heidelberg, pp. 39–66 (2006)
6. Grünbacher, P., Köszegi, S., Biffl, S.: Stakeholder Value Proposition Elicitation and Reconciliation. In: Value-Based Software Engineering, pp. 133–154 (2006)
7. Parkin, M.: Economics. Pearson Education, Boston (2008)
8. Briggs, R., Gruenbacher, P.: EasyWinWin: Managing Complexity in Requirements Negotiation with GSS. In: Proceedings of the 35th Annual Hawaii International Conference on System Sciences. IEEE Computer Society, Big Island (2002)
9. Oza, N., Biffl, S., Frühwirth, C., Selioukova, Y., Sarapisto, R.: Reducing the Risk of Misalignment between Software Process Improvement Initiatives and Stakeholder Values. In: Industrial Proceedings of EuroSPI 2008, Publizon, Dublin, pp. 6.9-6.18 (2008)
10. Keeney, R.L., Raiffa, H., Rajala, D.W.: Decisions with Multiple Objectives: Preferences and Value Trade-Offs. IEEE Transactions on Systems, Man, and Cybernetics 9, 403 (1979)
11. Brynjolfsson, E., Kemerer, C.F.: Network Externalities in Microcomputer Software: An Econometric Analysis of the Spreadsheet Market. Management Science 42, 1627–1647 (1996)

12. Gandal, N.: Competing Compatibility Standards and Network Externalities in the PC Software Market. The Review of Economics and Statistics 77, 599–608 (1995)
13. Katz, M.L., Shapiro, C.: Network Externalities, Competition, and Compatibility. The American Economic Review 75, 424–440 (1985)
14. Brandenburger, A.M., Nalebuff, B.J.: Co-Opetition: A Revolution Mindset That Combines Competition and Cooperation: The Game Theory Strategy That's Changing the Game of Business. Doubleday, New York (1996)
15. Metcalfe, B.: Metcalfe's Law: A network becomes more valuable as it reaches more users. InfoWorld 17, 53 (1995)
16. Gallaugher, J.M., Wang, Y.: Understanding Network Effects in Software Markets: Evidence from Web Server Pricing. MIS Quarterly 26, 303–327 (2002)
17. Kotler, P., Keller, K.L.: Marketing Management. Prentice Hall, Upper Saddle River (2006)
18. Nelson, P.: Information and Consumer Behavior. Journal of Political Economy 78, 311 (1970)
19. Messerschmitt, D.G., Szyperski, C.: Software Ecosystem: Understanding an Indispensable Technology and Industry. The MIT Press, Cambridge (2003)
20. Rohlfs, J.H.: Bandwagon effects in high-technology industries. MIT Press, Cambridge (2001)
21. Boztepe, S.: Toward a framework of product development for global markets: a user-value-based approach. Design Studies 28, 513–533 (2007)
22. Lee, Y., O'Connor, G.: New product launch strategy for network effects products. Journal of the Academy of Marketing Science 31, 241–255 (2003)
23. Dane, E., Pratt, M.G.: Exploring Intuition and Its Role in Managerial Decision Making. Academy of Management Review 32, 33–54 (2007)
24. Shapiro, C., Varian, H.R.: Information rules a strategic guide to the network economy. Harvard Business School Press, Boston (1999)
25. Klemperer, P.: Markets with Consumer Switching Costs. The Quarterly Journal of Economics 102, 375–394 (1987)
26. Arthur, W.B.: Competing Technologies, Increasing Returns, and Lock-In by Historical Events. The Economic Journal 99, 116–131 (1989)
27. Boehm, B.W., Ross, R.: Theory-W Software Project-Management - Principles and Examples. IEEE Transactions on Software Engineering 15, 902–916 (1989)
28. Bowman, C., Ambrosini, V.: Value creation versus value capture: Towards a coherent definition of value in strategy. British Journal of Management 11, 1–15 (2000)
29. Schilling, M.A.: Technological Lockout: An Integrative Model of the Economic and Strategic Factors Driving Technology Success and Failure. Academy of Management Review 23, 267–284 (1998)
30. Schilling, M.A.: Toward a General Modular Systems Theory and Its Application to Interfirm Product Modularity. The Academy of Management Review 25, 312–334 (2000)
31. Cowan, R., Foray, D.: The Economics of Codification and the Diffusion of Knowledge. Industrial & Corporate Change 6, 595–622 (1997)
32. Varian, H.R., Farrell, J., Shapiro, C.: The Economics of Information Technology: An Introduction. Cambridge University Press, Cambridge (2004)
33. Teece, D.J.: Technology and Technology Transfer: Mansfieldian Inspirations and Subsequent Developments. Journal of Technology Transfer 30, 17 (2005)
34. Quintas, P.: Programmed Innovation? Trajectories of Change in Software Development. Information Technology & People 7, 25–47 (1994)

35. Anderson, P., Tushman, M.L.: Technological Discontinuities and Dominant Designs - a Cyclical Model of Technological-Change. Administrative Science Quarterly 35, 604–633 (1990)
36. Vogel, H.L.: Entertainment Industry Economics: A Guide for Financial Analysis. Cambridge University Press, Cambridge (2001)
37. Abernathy, W.J., Clark, K.B.: Innovation: Mapping the winds of creative destruction. Research Policy 14, 3–22 (1985)
38. Baldwin, C.Y., Clark, K.B.: Design Rules: The Power of Modularity. MIT Press, Cambridge (2000)
39. Franke, N., Hippel, E.V.: Satisfying heterogeneous user needs via innovation toolkits: the case of Apache security software. Research Policy 32, 1199–1215 (2003)
40. Schilling, M.A.: Intraorganizational Technology. In: Baum, J.A.C. (ed.) Companion to Organizations, pp. 158–180. Blackwell Publishers, Malden (2002)
41. Nambisan, S.: Complementary product integration by high-technology new ventures: The role of initial technology strategy. Management Science 48, 382–398 (2002)
42. Sengupta, S., Sengupta, S.: Some Approaches to Complementary Product Strategy. Journal of Product Innovation Management 15, 352 (1998)
43. Banker, R.D., Datar, S.M., Kemerer, C.F., Zweig, D.: Software Complexity and Maintenance Costs. Communications of the ACM 36, 81–94 (1993)
44. Kemerer, C.: Software complexity and software maintenance: A survey of empirical research. Annals of Software Engineering 1, 1–22 (1995)
45. Lehman, M., Ramil, J., Wernick, P., Perry, D., Turski, W.: Metrics and laws of software evolution-the nineties view. In: Proceedings of the Fourth International Software Metrics Symposium, pp. 20–32. IEEE Computer Society, Albuquerque (1997)
46. Orton, J.D., Weick, K.E.: Loosely Coupled Systems: A Reconceptualization. The Academy of Management Review 15, 203–223 (1990)
47. Cusumano, M.A., Yoffie, D.B.: Competing on Internet Time – Lessons from Netscape and Its Battle with Microsoft. Free Press, New York (1998)
48. Helfat, C.E., Peteraf, M.A.: The dynamic resource-based view: Capability lifecycles. Strategic Management Journal 24, 997–1010 (2003)
49. Vessey, I.: The role of cognitive fit in the relationship between software comprehension and modification. MIS Quarterly 30, 29–55 (2006)
50. Zhang, J.: The nature of external representations in problem solving. Cognitive Science 21, 179–217 (1997)
51. Zhang, J., Norman, D.A.: Representations in distributed cognitive tasks. Cognitive Science 18, 87–122 (1994)
52. Basili, V.: The Experience Factory and its relationship to other Improvement Paradigms. In: Sommerville, I., Paul, M. (eds.) ESEC 1993. LNCS, vol. 717, pp. 68–83. Springer, Heidelberg (1993)