

Peter Buxmann
Heiner Diefenbach
Thomas Hess

The Software Industry

Economic Principles,
Strategies, Perspectives

 Springer

The Software Industry

Peter Buxmann · Heiner Diefenbach
Thomas Hess

The Software Industry

Economic Principles, Strategies,
Perspectives

Peter Buxmann
Chair of Information Systems/
Wirtschaftsinformatik
TU Darmstadt
Darmstadt
Germany

Thomas Hess
Institute for Information Systems
and New Media
LMU München
Munich
Germany

Heiner Diefenbach
TDS AG
Neckarsulm
Germany

ISBN 978-3-642-31509-1 ISBN 978-3-642-31510-7 (eBook)
DOI 10.1007/978-3-642-31510-7
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2012945103

© Springer-Verlag Berlin Heidelberg 2013

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Foreword

Barely any other industry has changed society and the business world that lasting as software industry. Software support of business processes inside and between companies has become a matter of course as to “google” for information or the use of navigation systems. So, it is not surprising that the software industry is one of the biggest growth markets worldwide and also one of the most international sectors. Accordingly, software companies compete for customers and to an increasing degree for employees all over the world. But not only due to the international software markets, but also as a result of the unique characteristics of software as a product, there are special rules for software providers. That is the issue of this book.

This book provides a clear structure: The first part discusses general principles of the software industry, while the second deals with specific topics. The first chapter describes the beginnings of the software industry and its basic rules. We discuss the economic principles of the software industry in [Chap. 2](#) as the properties of digital goods, the network effects on software markets, the issue of standardization, and look also at the aspects of transaction cost and agency theory. Against this background we examine selected strategies for software vendors. The second part highlights approaches to outsourcing by software providers and users as well as platform concepts. Moreover, the last chapters are looking at software as a service and open source software.

We would like to express our sincere gratitude to our colleagues at Software Economics Group Darmstadt-München (www.software-economics.org) for their assistance and support. In particular, Prof. Dr. Alexander Benlian, Dr. Björn Brandt, Christoph Burkard, Tobias Draisbach, Jin Gerlach, Thomas Görge, Daniel Hilkert, Christian Hörndlein, Dr. Sonja Lehmann, Dr. Janina Matz, and Dr. Christian Wolf contributed new content, taken in part from their doctoral theses.

We would like to thank Anette von Ahsen and Helena Wenninger for their assistance with the preparation of this book. Not only did they provide us with numerous bug reports, but also the corresponding patches. We are equally thankful to Dr. Niels Thomas and Alice Blanck from our publishers, Springer. Again, it was a pleasure to work with them.

We look forward to your feedback and comments and hope that you will find the book stimulating and enjoyable.

April 2012

Peter Buxmann
Heiner Diefenbach
Thomas Hess

Contents

Part I Basics

1	The Rules in the Software Industry	3
1.1	Software and Software Markets: Unique Characteristics of the Software Industry	3
1.2	The Beginnings of the Software Industry	4
1.3	Types of Software Provider and Users' Selection Criteria	4
1.3.1	Software Providers in the Wider and Narrower Sense	4
1.3.2	The Selection of Software	9
1.4	Business Models for Software Companies	14
1.5	Revenue from Services in the Software Industry	16
2	Economic Principles in the Software Industry	19
2.1	Properties of Digital Goods	19
2.2	Network Effects on Software Markets: The Winner Takes it All	20
2.2.1	Network Effects: Basics and Definitions	21
2.2.2	Impact of Network Effects on Software Markets	23
2.2.3	Structure of Software Markets	26
2.2.4	Network Effects as a Competitive Factor	27
2.2.5	A Case Study: Two-Sided Network Effects and Platform Strategies in the Digital Games Industry	29
2.2.6	Limitations of Network Effect Theory	32
2.3	The Standardization Problem	33
2.3.1	Approach and Background	33
2.3.2	The Central Standardization Problem as An Optimization Problem	36
2.3.3	The Decentralized Standardization Problem: A Game Theoretical Approach	37

2.3.4	The Standardization Problem: Lessons Learned	40
2.4	Transaction Cost Theory: In Search of the Software Firm's Boundaries	41
2.4.1	Starting Point and Elements of Transaction Cost Theory	42
2.4.2	Division of Labor Among Companies: A Transaction Cost Theory Perspective	44
2.4.3	Structural Changes to Transaction Costs: The Move to the Middle	45
2.4.4	Excursion: Intermediaries and Transaction Costs	46
2.5	Software Development as a Principal-Agent Problem.	47
2.5.1	Incentive-Compatible Compensation and Efficient Control.	47
2.5.2	Principal-Agent Relationships: Definitions and Basic Principles	48
2.5.3	Incentive-Compatible Compensation Schemes	49
2.5.4	Control Systems	52
3	Software Vendor Strategies	55
3.1	Cooperation and Acquisition Strategies	55
3.1.1	Cooperation in the Software Industry	55
3.1.2	Mergers and Acquisitions in the Software Industry	64
3.2	Sales Strategies	71
3.2.1	Structuring of Sales Systems: Organization and Sales Channels in the Software Industry.	71
3.2.2	Organization of Relationships with Sales Partners and Key Accounts.	75
3.2.3	Key Performance Indicators as a Sales Performance Management Tool in the Software Industry	77
3.3	Pricing Strategies.	81
3.3.1	Background	81
3.3.2	Pricing Models for Software Products	82
3.3.3	Pricing Strategies of Software Providers: Empirical Findings	96
3.3.4	Approaches to Pricing for Custom Software Providers.	99
3.4	Development Strategies	101
3.4.1	Structuring of the Software Development Process.	101
3.4.2	Software-Supported Software Development	105
3.4.3	HR Management in Software Development	107

Part II Specific Issues

4 Outsourcing and Offshoring of Software Development.	113
4.1 Overview	113
4.2 Forms of Outsourcing and Offshoring	114
4.3 Motives for Outsourcing and Offshoring	117
4.3.1 Cost Savings	117
4.3.2 Greater Flexibility	119
4.3.3 Concentration on Core Competencies	119
4.3.4 Acquisition of Knowledge and Skills	119
4.3.5 Exploitation of the “Follow-the-Sun” Principle	120
4.4 Selection of Locations by Software Providers	120
4.5 Outsourcing by Software User Organizations	123
4.5.1 Outsourcing of the Development of New Custom Software	123
4.5.2 Outsourcing Modifications to Standard Software	126
4.5.3 Outsourcing the Further Development and Maintenance of Application Software	129
4.5.4 User Satisfaction with Onshore, Nearshore, and Farshore Providers	133
4.6 Nearshoring Versus Farshoring: Distance from the Customer as a Success Factor?	134
4.6.1 Cultural and Language Barriers in Offshore Projects	134
4.6.2 The Importance of Face-to-Face Meetings to Project Success	136
4.6.3 Time Difference: Challenges and Opportunities	137
4.7 Outsourcing by Software Providers	139
4.7.1 The Status Quo of Specialization and the Division of Labor: Insights from Three Case Studies	139
4.7.2 Future Division of Labor in the Software Industry	146
5 Platform Concepts	155
5.1 Overview	155
5.2 Product Platforms in the Software Industry	155
5.2.1 Cost Structure of Platform-Based Software Development	155
5.2.2 Organizational Support for Implementing Product Platforms	159
5.2.3 Add-on: Industrialization as a Management Concept for the Software Industry	159
5.3 Industry Platforms in the Software Industry	162
5.3.1 Openness of Industry Platforms	162
5.3.2 Management of Complementors	165

6	Software as a Service: The Application Level of Cloud Computing	169
6.1	Overview	169
6.2	Basic Principles of Cloud Computing.	170
6.3	SaaS: Applications and Examples	174
6.4	SaaS from the User's Perspective: Opportunities and Risks	176
6.4.1	Background	176
6.4.2	Empirical Study on Opportunities and Risks for SaaS Users	180
6.5	SaaS from the Provider's Perspective: Pricing Strategies and Business Models	183
6.5.1	Basic Considerations	183
6.5.2	Empirical Study of SaaS Providers' Pricing Strategies and Business Models	184
6.5.3	Case Study to Compare Usage-Based and Usage-Independent Pricing Models	187
7	Open Source Software	191
7.1	Overview	191
7.2	Features of Open Source Software.	191
7.3	Open Source Projects: Principles and Motivation of Software Developers	196
7.3.1	Organizational Structures and Processes in Open Source Projects.	196
7.3.2	Contributor Motivation	197
7.4	Open Source Software: The User Perspective	199
7.5	Commercial Software Vendors' Involvement	200
7.6	Open Source ERP Systems	202
	References	211
	Index	221

Abbreviations

ASD	Adaptive software development
ASP	Application service providing
BOT	Build operate transfer
BPEL	Business process execution language
BSA	Business software alliance
BSD	Berkeley software distribution
B2B	Business-to-business
B2C	Business-to-company
CAR	Cumulative abnormal return
CAAR	Cumulative average abnormal return
CASE	Computer aided software engineering
CCC	Content communication and collaboration
CEO	Chief executive officer
CIM	Computational independent model
CIO	Chief information officer
CMMI	Capability maturity model integration
CORBA	Common object request broker architecture
CPU	Central processing unit
CRM	Customer relationship management
CTO	Chief technology officer
CVS	Concurrent versions system
DCOM	Distributed component object model
EDI	Electronic document interchange
EPK	Event-driven process chain
ERP	Enterprise resource planning
GNU	GNU is not Unix
GPL	General public license
HTTP	Hypertext transfer protocol
ICT	Information and communication technology
IE	Internet explorer
IP	Intellectual property
ISO	International organization for standardization
IT	Information technology
ITT	Indian Institute of technology

JDK	Java development kit
KPI	Key performance indicator
LGPL	Library/Lesser general public license
LoC	Lines of code
M&A	Mergers & Acquisitions
MDE	Model driven engineering
MIT	Massachusetts institute of technology
MPL	Mozilla public license
MU	Monetary units
ODF	Open document format
OEM	Original equipment manufacturer
OSI	Open source initiative
OSS	Open source software
PDF	Portable document format
PIM	Platform independent model
PM	Platform model
PMC	Point of marginal cheapness
PME	Point of marginal expensiveness
PPC	Production planning and control
PSM	Platform specific model
PSM	Price sensitivity meter
ROI	Return on investments
RPC	Remote procedure call
R&D	Research and development
SaaS	Software as a service
SIIA	Software and Information Industry Association
SCM	Supply chain management
SME	Small and mid-size enterprises
SOA	Service-oriented architecture
SQL	Structured query language
SW	Software
TIME	Telecommunications, information, media and entertainment
UDDI	Universal description, discovery, and integration
URI	Uniform resource identifier
US-GAAP	United States Generally Accepted Accounting Principles
VHS	Video home system
WSDL	Web services description language
W3C	World wide web consortium
WTP	Willingness to pay
XML	Extensible markup language
XP	eXtreme programming

Part I Basics

1.1 Software and Software Markets: Unique Characteristics of the Software Industry

The software industry is fundamentally different from other industries. This is partly due to the unique nature of software as a product, but also the structure of software markets.

A distinctive feature of *software products* is that they, like all other digital goods, can be reproduced cheaply. In other words, variable costs are close to zero. This cost structure has the result that the licensing side of software providers' business is generally more profitable than the service side—or at least it appears to be, something we will discuss in more detail later. Moreover, software can be copied any number of times without loss of quality. Once a copy is available on the Internet, intellectual property rights are practically unenforceable. This especially applies to easy-to-understand products on business-to-consumer markets. In addition, once a software product has been developed, it is relatively simple to create different versions or packages and sell these to separate groups of customers at different prices.

Software markets also have some unique characteristics. The software industry is more international in nature than practically any other sector. Software can be developed by distributed teams working almost anywhere in the world, and sold over the Internet in seconds, at negligible cost. This has fueled global competition between software providers. In comparison with other industries, providers in many segments enjoy little “home advantage” in their national markets. Moreover, the network effects associated with software often creates winner-takes-all markets. This accounts for the large number of mergers and acquisitions, for example.

These and other specific economic principles and game rules will be discussed later in more detail. After all, they are the backdrop against which software industry players formulate their strategies and business models.

But first, we wish to briefly outline the historical development of the industry.

1.2 The Beginnings of the Software Industry

The software industry is relatively young. Its origins date back to the early 1950s, when it was still common practice to sell software and hardware as a single package. At that time, software was an integral part of hardware, and was still referred to as program code—the term software was first used in 1959 (Campbell-Kelly 1995, p. 80). In the USA, this period saw the establishment of the first small companies to develop software within the scope of specific projects (Hoch et al. 2000, p. 27 ff).

The profile of software rose in 1969, when the US Department of Justice demanded that IBM itemize hardware and software separately on its invoices. During the 1960s, a whole series of companies that focused exclusively on software development sprang up. Of these, Microsoft is most worthy of mention: founded jointly by Bill Gates and Paul Allen, the company began developing programming languages—first BASIC, and later others such as FORTRAN and COBOL—for various processors and computers. Later, in collaboration with IBM, Microsoft developed MS-DOS, which would become the standard operating system, significantly contributing to the spread of the personal computer (Ichbiah 1993, pp. 91–116). Later, the company would offer applications as well, entering into competition with Lotus and the like. As early as 1983, Bill Gates told in *Business Week* that it was Microsoft’s goal to become a one-stop provider of all types of PC software (Ichbiah 1993, p. 141).

At the same time as Microsoft embarked on its long rise, another success story was unfolding in the small town of Walldorf, Germany. Dietmar Hopp, Hans-Werner Hector, Hasso Plattner, Klaus Tschira, and Claus Wellenreuther established a company specializing in the development of software for business applications and processes: SAP AG was born. At first, it developed software for mainframes, later offering applications for client–server environments. Today, SAP is the largest European software company and the world leader in enterprise resource planning (ERP) systems.

These first two examples already reveal a key characteristic of the software industry: often, just one technology or vendor will come to dominate the market. For example, MS-DOS squeezed out the rival CPM operating system, while the Excel spreadsheet program trounced competing products such as Lotus 1–2–3. Microsoft was to become the world’s leading provider of office applications, browsers, and operating systems. The market for business software is currently undergoing consolidation (see [Sect. 3.1.2](#)). We will return to the special features of these markets—but will now turn our focus to the key players in the software industry.

1.3 Types of Software Provider and Users’ Selection Criteria

1.3.1 Software Providers in the Wider and Narrower Sense

In the following section, we will introduce the various types of software providers. A distinction should be made between software providers in the broader and narrower sense.

The role of a *software provider in the narrower sense* is to develop software—and this applies to all types of software. Software itself can be categorized in various ways. A commonly used criterion is how directly it interacts with the hardware (i.e., at what level of the system stack does it operate. According to this method, software comprises system software (e.g., operating systems), middleware at the next level up, and application software (e.g., for word processing or accounting) above that. Software can also be divided into products for commercial and for private users. A third criterion for classification of software, and one that is more pertinent to our discussion, is the degree of standardization. On this scale, custom software and standard software are at the end points.

Custom software is developed, in line with specific customer requirements. It is developed either in-house—generally in the IT department, but in some cases, in relevant user departments—or by an external software vendor. In India particularly, the custom software development industry is booming. A number of successful players with enormous growth rates have emerged, such as Tata Consultancy Services, Wipro Technologies, Infosys Technologies, and Cognizant Technology Solutions. In addition, there are multitudes of smaller or mid-size companies that are still operating nationally or regionally in high-wage countries. By contrast, custom software development companies based in low-wage countries are more likely to operate globally, and so compete against one another. We will examine these nearshore and farshore providers and their impact on the software industry more closely in [Sect. 4.6](#).

Standard software is generally developed for the mass market. As a result, providers address the lowest common denominator in terms of users' needs. In the following pages, we will analyze the rise of, and success factors behind, standard software, using SAP as a case study.

SAP and its R system

SAP is the leading provider of business software in Europe and the fourth largest independent software vendor in the world. The company employs 55,000 people at 75 subsidiaries worldwide. Overall 183,000 customers reap the benefits of SAP's solutions.

Key factors

SAP's success is attributed to three key factors; namely, the idea of standard software, offering integrated solutions, and real-time processing. In 1981, the company put these ideas into practice for the first time with SAP R/2, thereby opting to focus on the development of business software. SAP launched the R/3 system a decade later. Originally, the new architecture was not meant to replace R/2 but complement it by providing a solution for SMEs. However, the general changeover from mainframe solutions to the client-server model, which was underway at that time, made R/3 just as appealing to larger companies and led to its huge popularity. Since then, SAP has extended its product portfolio. We will return to this topic later.

Standard software

In the early days, as mentioned above, software was generally developed within customer-specific projects. This meant that the software and, if necessary, the hardware was tailored exactly to the customer's requirements. By contrast, SAP planned from the outset to develop a standardized system that could be used by multiple customers. SAP was one of the first companies to systematically pursue this approach.

An integrated solution

At the heart of the SAP system is an integrated database that all applications can access. Using an integrated data repository was a completely new approach in the 1970s. At the time, redundant and inconsistent data generated enormous costs. Building on this foundation, SAP gradually developed modules for various corporate functions, such as financial and management accounting, logistics, and human resource management. Initially, these packages were primarily designed for industrial companies and could be purchased individually or as a package which included a database.

In contrast to office applications such as those offered by Microsoft, it became clear that many areas of business had to address industry-specific requirements. To meet this need, SAP now offers a wide variety of industry-specific solutions. This included the service sector, which had received little attention up to that point.

Real-time data processing

By the end of the 1970s, companies generally used punch cards, i.e., data were entered into the computer via punch cards and not processed until later. From the outset, SAP's founders saw real-time processing as a key feature, which was implemented in all systems. This also explains the "R" in the product name, which stands for real time.

Sources Leimbach (2007) Vom Programmierbüro zum globalen Softwareproduzenten. Die Erfolgsfaktoren der SAP von der Gründung bis zum R/3-Boom. In: Zeitschrift für Unternehmensgeschichte 52: 5–34;

www.sap.com.

There is no clear distinction between custom and standard software's. Even standard business software can be customized to users' needs to a certain extent. Nonetheless, implementation projects involving software parameterization and/or customization, and, where required, function extensions, frequently cost into the millions. So, it generally makes sense for users to opt for smaller adjustments only: complete modification of standard software, until it is 100 % geared to users' needs, is extremely expensive; it can also create problems when upgrading to new versions. In addition, new approaches, such as service-oriented architectures, offer the possibility (at least theoretically) of selecting software that best matches their

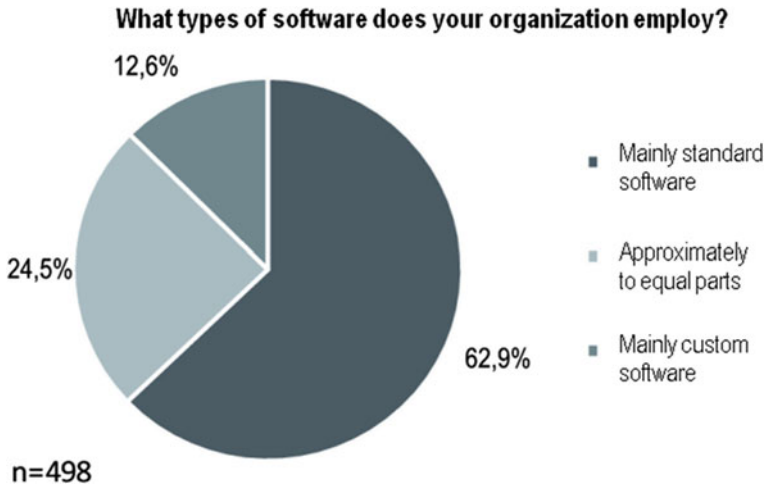


Fig. 1.1 Share of software types

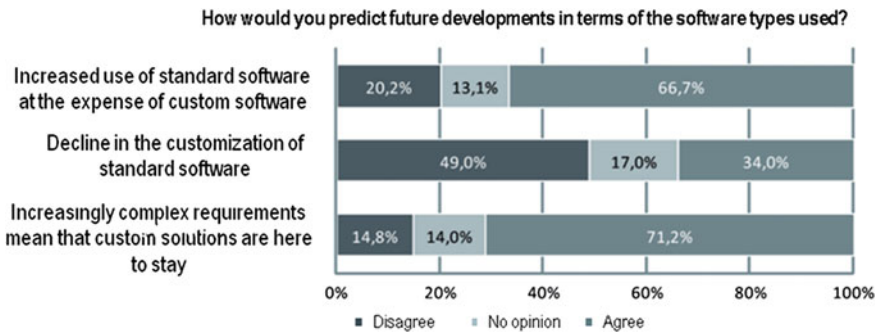


Fig. 1.2 Predicted developments in the share of software types used

requirements in specific areas, and then melding them into a single customized application solution by means of integration software (see Sect. 4.7.2).

The proportion of standardized software solutions in enterprises' portfolios is set to increase, as our survey of 498 German CIOs shows (Buxmann et al. 2010). Total 62.9 % of respondents' companies mostly use standard software solutions, while 24.5 % employ standard and custom softwares more or less equally. Custom software dominates at 12.6 % of enterprises (Fig. 1.1).

In addition to asking about their companies' current use of standard and custom softwares, we also canvassed respondents' opinions as to the future development of these types of software. Almost 66.7 % of the CIOs surveyed agreed with the statement that companies are increasingly employing standard software at the expense of custom software. Out of this 20.2 of companies disagreed, while 13.1 % expressed no opinion (see Fig. 1.2). Total 34 % of the participants

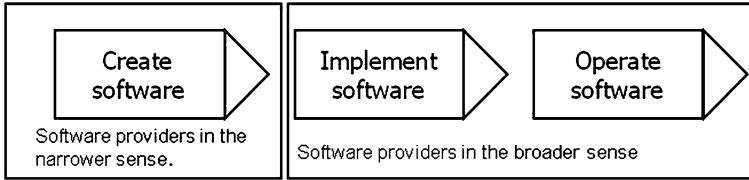


Fig. 1.3 Classification of software providers

concluded with the assertion that customizations of standard software would decrease, while almost half (49 %) disputed this assumption. The overwhelming majority of respondents (71.2 %) believed that, due to growing user requirements, custom solutions would continue to be needed sometimes to address specific problems, and would have to be integrated into the overall IT landscape.

In future, therefore, companies will employ more standard software solutions; however, customizations will still be necessary. For specific problems, for which no standard application software is available (as yet), companies will continue to develop individual solutions (either in-house or via outsourcing), and then have to integrate them into their IT environments.

In addition to software providers in the narrower sense—in other words, companies that develop standard or custom software—there are also those that offer services for the later phases in the lifecycle of software solutions. In the following, we have described these organizations as software providers in the broader sense. Their services include both implementation support and ongoing operation. Figure 1.3 depicts our classification of software providers.

Especially, in the case of complex software solutions that are not self-explanatory—in other words, those that are not easy to implement and integrate into an application environment—there is currently great demand for services. As a result, a large number of providers are active in this space. These can be broken down as follows (Lünendonk 2009):

- IT consultancies and system integrators (which offer IT consulting, development of custom software, etc.),
- IT service providers (outsourcing, ASP, user training etc.) and
- Business innovation/transformation partners (management and IT consulting, system implementation).

The provision of user support for SAP implementation projects has traditionally been a significant market. This is because—as discussed above—these projects involve customizing the software to the specific needs of the user organization. In many cases, companies using SAP software do not have the necessary knowledge and skills, or have insufficient employees with this expertise. Such implementation projects are crucial for the customer to gain the full benefits from the software, as it directly affects internal and inter-company processes. In addition, these projects frequently entail minor programming tasks, such as the development of interfaces between heterogeneous systems.

This business relies like almost no other on a relationship of trust between customer and service provider. Detlev Hoch et al. even describe this relationship in terms of “faith”. In their view, the customer must have faith in the outstanding skills of the service provider or the employees assigned to the project, and in their ability to solve the customer’s problems as promised (Hoch et al. 2000, p. 160). While users can test-drive products from a standard software vendor, this is simply not possible in the case of bespoke solutions. Often, the customers have no alternative, but to base their decision on the references furnished by the consultants and system integrators. Against this background, these service providers’ marketing activities have the primary goal of building trust in their skills and ability to deliver. To this end, the following activities are essential: (Hoch et al. 2000, pp. 162–178):

- Sponsoring IT conferences,
- Discussion groups and forums with leading lights from the IT and software industries,
- Publications in industry media and scientific journals and
- Advertisements and TV commercials.

For customers, the complexity of implementation and integration projects means that selecting the right service provider is crucial. Especially, considering that most projects go over budget—and in many cases, fail altogether. In a study by Standish Group International Inc, just 32 % of the IT projects in their sample were successfully completed within the allotted time and budget (The Standish Group International 2009, p. 2). Overall 44 % far exceeded these targets, while 24 % were never completed at all. Projects assessed as being complex exceeded their budget and planned duration by more than 100 %.

Operating IT solutions has been a key part of the outsourcing business for many years (also see [Chap. 4](#)). There are a large number of players on this international and highly competitive market. We include them among our software providers in the broader sense.

In recent years, a new form of IT delivery has emerged: cloud computing, which will be examined in [Chap. 5](#).

1.3.2 The Selection of Software

The business models of software providers (both in the narrower and broader sense) reflect the selection mechanisms and preferences of their customers. The following sections will outline the processes and criteria used for evaluating and selecting software solutions by businesses ([Sect. 1.3.2.1](#)) and consumers ([Sect. 1.3.2.2](#)).

1.3.2.1 Software Selection by Businesses

Companies generally go through several sequential steps to reach their final choice (see [Fig. 1.4](#)). This involves gradually reducing the choice of possible alternatives.

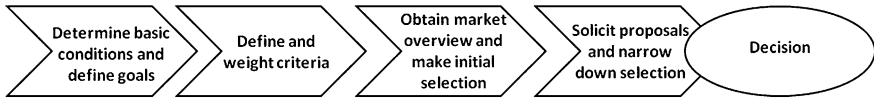


Fig. 1.4 Software selection by companies

Typical context variables are company-related factors, such as industry, size or business functions to be supported, or environmental variables, such as technological standards. These basic conditions are used to define the goals which purchasing the software must achieve. The next step is usually to define criteria. These are characteristics that describe a software system and that form the basis for evaluation.

Market overviews are used to identify potential options. Typically, this is followed by a two-stage evaluation and selection process. In the first round, the options are assessed according to elimination criteria. Only solutions that meet these criteria are examined in more detail. With the help of a detailed definition of requirements (functional specifications), which is derived from the criteria, the company will then request quotations or additional information on the remaining options. Based on this information, a further round of selection follows, culminating in a selection recommendation. Some approaches also include post-selection steps, such as contract negotiations and functional testing to estimate the customization effort required.

The following section explores the development of a system of goals, the selection and weighting of criteria, and the activities involved in the two selection rounds in more detail.

The formulation of goals that are to be met by introducing or deploying software in a particular area is generally the starting point for analysis, and provides a yardstick for assessing alternatives. A variety of methods can be used; for example, goals can be derived from the top down, i.e., from corporate goals or the IT strategy. For example, high-level corporate goals, such as increasing profits, can be applied to the concrete problem of selecting standard software for a specific purpose. One way of doing this is to breakdown the high-level goals into concrete subordinate goals in terms of an end-means relationship with respect to the selection problem. Examples include reducing throughout, cutting inventory levels, or improving response times.

A bottom-up approach is also possible. The goals derived in this way have a very functional focus with respect to the specific selection problem and/or to existing structures (such as the IT architecture). Examples include goals in the form of general characteristics of software, such as independence from database or hardware vendors, compatibility with existing versions, or specific application-related functionality, such as automatic allocation of part numbers with checksum or broadening the information base. These characteristics often correspond to concrete requirements rather than being goals in the narrow sense, which are pursued for their own sake. In practice, to determine the goals needed as a basis for the criteria, most companies employ a combination of top-down and bottom-up methods.

Table 1.1 Typical selection criteria in the selection of ERP systems

Selection criteria (software)	Selection criteria (implementation/provider)
Functionality	Ability of software to be integrated into existing IT architecture
Cost	Implementation time/Customization costs
User friendliness	Support from provider
Reliability	Reputation of provider

If criteria are derived from the goal system, they generally represent the lowest level of this system. As a result, the effect of a given system property on the desired goals is clearly visible. Selection criteria are often subdivided into software, implementation, and provider-related properties to reflect the goal hierarchy. Besides the characteristics of the software itself, implementation criteria also play a key role, especially in complex systems. This is because over and above the procurement costs, implementation work can significantly impact the software's total cost. The implementation of complex software systems is often carried out with the support of software providers or implementation partners, and generally speaking, their expertise and professionalism should also be evaluated before selecting software. In addition to purely rational criteria, the role of political factors (e.g., informal agreements, nepotism, and internal power struggles) should not be underestimated. These can have a significant influence on the selection process (Howcroft and Light 2006).

Table 1.1 shows which selection criteria for software, implementation, and providers are actually used in practice in the selection of ERP systems (Keil and Tiwana 2006; Jadhav and Sonar 2009).

In terms of weighting criteria toward the overall decision, it is generally assumed that these criteria are already, or will be fulfilled. However, it is also possible for a criterion to be partially met. In this case, it is necessary to specify a desired goal threshold or evaluate various thresholds. For example, a criterion dubbed "database independence" does not make it clear whether compatibility with two, three, or ten database management systems is required. This can be important to ensure solutions are not overrated, for example, where the threshold is set relatively low. With some goals, on the other hand, a certain minimum level needs to be met before an option will even be considered or investigated further. Such knockout (or killer) criteria make the selection process more efficient. However, they also pose the risk of ruling out options that only narrowly miss the minimum requirement.

A holistic approach is usually employed to weight criteria, for example, the kind familiar from cost-benefit analyses. Weightings are generally allocated by a direct comparison of criteria. For example, empirical studies on the selection of ERP systems have shown that functionality, reliability, and cost are weighted more heavily, while user friendliness and implementation effort are regarded as less important (Keil and Tiwana 2006). However, problems arise with this approach when a highly differentiated system of weightings is used. Especially with

complex selection problems, which owing to the large number of evaluation criteria, include the selection of software, it is only possible to rank options according to an ordinal scale. More precise scores imply (pseudo-) accuracies that seldom reflect with the decision maker's actual preferences.

In practice, drawing up market overviews, shortlisting and evaluating alternatives are often interdependent tasks that are carried out at the same time. Whether or not a system is included in a market overview is a form of selection in itself. Due to the wide array of offerings available, this first selection step is commonly shaped by heuristic approaches such as:

- Inclusion of systems employed by competitors,
- Inclusion of systems currently receiving widespread coverage in industry media (e.g., Google Apps Premier, SAP ERP),
- Inclusion of systems that employees already have experience with,
- Inclusion of systems that, if selected, the consultant hopes will lead to a follow-up contract at the implementation stage and
- Inclusion of systems based on a random selection, e.g., through a visit to a trade show.

As explained above, evaluation and selection are an interconnected process, in which knockout criteria can reduce the number of alternatives. The following methods are typically employed:

- Evaluation of system descriptions,
- Evaluation of potential providers' bids in response to tender documents or requirements catalogs and
- Presentation of the system and/or concrete functionality based on defined application scenarios.

To evaluate certain criteria, a three-point scale comprising the values "fulfilled", "partially fulfilled", and "not fulfilled" is frequently used. Finally, individual evaluations need to be aggregated to form an overall assessment.

A system's suitability does not depend on how closely it meets functional requirements, but also on its costs. Accordingly, cost factors are often incorporated into the evaluation process by way of a cost-benefit coefficient. Ultimately, the software selection decision is made by comparing relevant quantitative and qualitative factors.

1.3.2.2 Software Selection by Consumers

While selection in companies is usually a group decision, consumers mostly make a choice alone. Much the same as in a business context, consumers typically go through four phases when selecting any product, including software (Blackwell et al. 2003). These are shown in Fig. 1.5.

Once a consumer has become aware of the need to buy new software (Phase 1: Determination of need), he or she will look for suitable information on available products that could satisfy the need. In doing so, the consumer, more or less

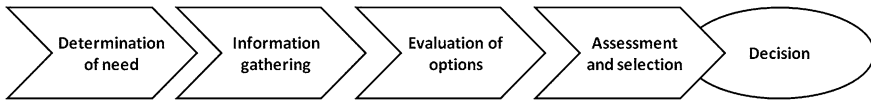


Fig. 1.5 Software selection process by consumers

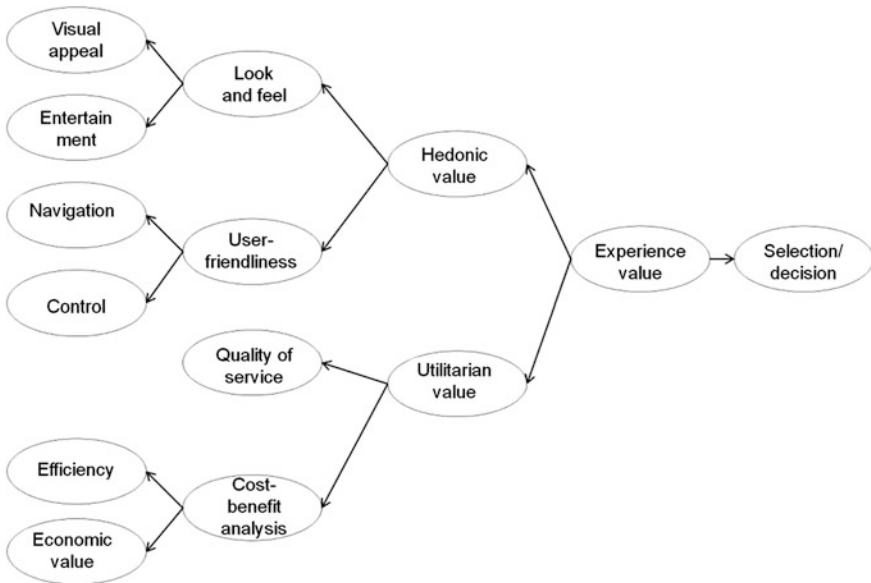


Fig. 1.6 Factors influencing the experience values of digital goods (following Mathwick et al. 2001)

consciously, identifies key subjective and objective criteria for decision making (Phase 2: Information gathering). After information gathering, the consumer processes the available data and evaluates the options in light of his or her personal selection criteria (Phase 3: Evaluation of options). Finally, the consumer compares and evaluates the available options and decides which product to purchase (Phase 4: Assessment and selection).

In contrast to selection decisions made by enterprises, consumers' choices are influenced not only by purely objective criteria (such as cost or functionality), but also individual or internal, i.e., personal (e.g., their personality, emotions, impressions, or lifestyle) and situational or external factors (e.g., culture or mouth-to-mouth propaganda). To help providers better understand which key criteria consumers use to make their selection, these factors will be examined in more detail below.

As mentioned above, in addition to purely rational selection criteria, soft factors play an important role in consumer decision making. A comprehensive model, which aptly reflects the special characteristics of digital products, is that of the experience value of digital goods as described by Mathwick et al. 2001 (see Fig. 1.6). It reveals

that digital products (e.g. computer games) have an experience value that is (consciously or unconsciously) evaluated before purchase. The experiential value is a combination of utilitarian and hedonic factors.

Utilitarian factors are overwhelmingly objective in nature and typically reflect cost-benefit calculations. This cost-benefit mindset, in turn, is fed by mainly rational evaluation criteria, such as functionality, compatibility, or the software's procurement costs. In addition, there is the service provided with the software, including the support, and guarantees (e. g., a hotline or warranty) the user will obtain on purchasing the software.

By contrast, *hedonic factors* appeal to the personal and situational needs of customers. Consumers' esthetic preferences play a key role, as does user friendliness. Both factors include more concrete subcriteria, e. g., visual attractiveness (such as colors and 3-D effects) or the way the software is navigated (e. g., mouse, keyboard, and joystick), which address consumers' specific personal purchasing motives.

Put together, these subcriteria influence the product's hedonic and utilitarian factors and so the entire experiential value of the software, which in turn impacts the consumer's software selection and ultimate purchase. These subcriteria can be influenced to a lesser or greater extent by software providers. The design of business models for software vendors is the subject of the following section.

1.4 Business Models for Software Companies

A business model describes the essential decisions that influence the economic performance of a company (cf. Osterwalder 2004). It outlines the products or services offered by the company as well as its revenue and distribution model. Furthermore, it addresses the embedding of a company in the value chain.

In the software industry, it all begun with companies that develop software by direct order of their customer. Such a software company gets in exchange directly paid by the ordering customer, either a fixed sum for the product or accordingly for the employed resources. The business model of the individual software manufacturer was born.

Relatively fast it became clear, that some software could be used in a variety of companies. The idea of standard software was formed, first for mainly system software but soon also for application software. It led to the business model of the standard software manufacturer. In this model, software is no longer developed for a specific customer—but the software now is developed for an anonymous market. However, this does not exclude pilot projects in individual companies. Well-known suppliers in this segment are Oracle (for system software) and SAP (for application software). In this model, a user company typically buys a license to use the software. The price depends on the numbers of installations or other indicators for usage of the software. Often the license fee also includes costs for maintenance and incremental upgrades of the software. Up to now, the user companies use software mainly to support their processes. Increasingly, however,

Table 1.2 Three generic business models of software companies

	Supplier of Individual Software	Supplier of Standard Software B2B	Supplier of Standard Software B2C
Offering	Software, developed for the specific needs of a company	Software, developed for a mass market on the business side	Software, developed for a mass market on the consumer side
Revenue Model	Payment for completed solution or used resources	One-time license fee plus maintenance fee	One-time license fee
Distribution Model	Direct contact with customers	Direct contact with customers	Digital or physical retail or in exceptions preinstallation on computer hardware

the software becomes part of the company's product offering (e.g., in cyber-physical systems of cars).

In the 1990s, the rising demand of private customers (B2C) enlarged the existing customer base (B2B) to a sizeable extend. Suppliers of standard software begun to serve this market also, but employed a modified business model than in B2B. Typically in this model, the private customer pays an up-front fee for an unlimited usage license of the software and receives incrementally upgraded versions as well. Distribution takes place directly through the Internet or indirectly through computer hardware retail shops, in exceptions also through the suppliers own brick and mortar shop.

Table 1.2 shows the three traditional business models (see also Valtakosi/Rönkkö 2009) once again in an overview.

Two specific models have to be taken in account. First, an attractive model for software companies is the preinstallation of software on end-user devices. This enables the supplier to gain a revenue share from the sold bundle of hardware and software. Bundles like this are offered to private customers. A second special case occurs when the software can be purchased over a platform which is located between the software company and the user. This setting becomes problematic for a software company if the platform is the only possible way to install the software on the end-user device and the company is not the platform provider. Such a position was established by Apple for their temporarily market dominating Smartphone, the iPhone.

Suppliers of software were often keen to extend their share of the value chain by offering complementary services to their products. In the case of suppliers of individual software solutions, this is often directly part of the customer's order. Suppliers of complex standard software in the B2B market offer sometimes operational or technical services that support the implementation of the software in the customer's organization. Therefore, they stand in partial competition with specialized service providers.

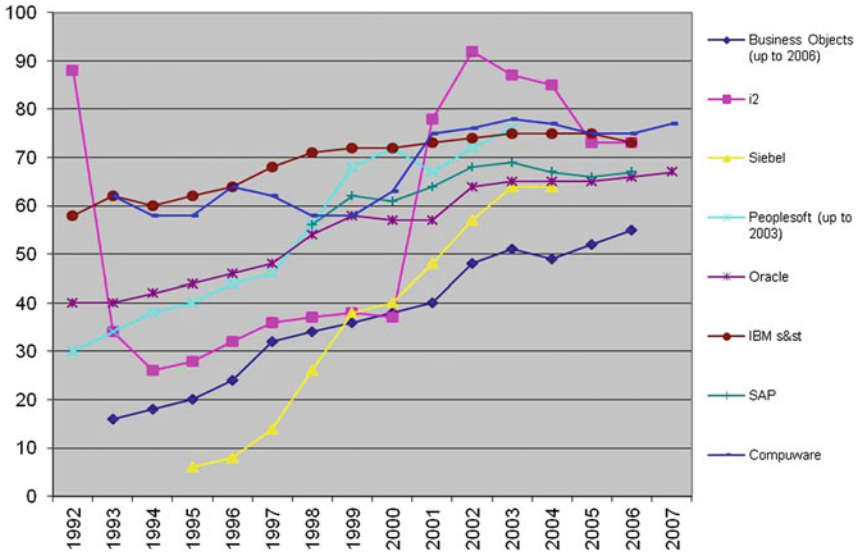


Fig. 1.7 Share of total sales generated by services, large standard software vendors (own calculations, building on Cusumano 2004, p. 37)

Equally, strong growth can be observed in the business customers' demand for the provisioning of services for data processing center. Specialized service providers in this segment are, e.g., T-Systems or IBM, but suppliers of software also try to enter in this business model. Recently, they started to offer Software-as-a-Service over the Internet and simplify the company-specific adjustments of software through, e.g., easy to use toolkits. SAP is currently developing their offers into this direction. Suppliers of that kind reach their turnover usually through charging monthly usage fees.

1.5 Revenue from Services in the Software Industry

In addition to incomes from licenses, software providers increasingly earn revenues from services. This is a source of revenue not just for custom software developers and software providers in the broader sense. It also offers standard software providers a wealth of opportunities to generate sales through services such as consulting and maintenance.

While consulting fees are commonly based on working days (and occasionally, on results), maintenance services generally involve the user paying the provider a percentage of the licensing fees annually. This percentage varies from provider-to-provider—a typical amount would be around 20 %/year. That makes it clear just how important this source of revenue is for software providers: If we assume that a software solution is used for an average 7–10 years before, it is replaced by a new

version—or, less commonly, a different product—we can see that the revenues from maintenance generally exceed those from licensing; this applies even after discounting revenues. Furthermore, this business model also ensures a regular income from maintenance fees, in contrast to licensing fees, providing a relatively steady source of revenue that software providers can factor into their planning. Figure 1.7 shows that, over the last few years, most major standard software vendors were able to significantly increase the share of their total sales generated by services.

But although revenue from services is significantly higher than income from licensing, this does not mean that providing services is necessarily a more attractive business than selling licenses. This is shown by comparing the profitability of the various sources of revenue. Licensing generally makes a 90–100 % profit (Cusumano 2004, p. 43 ff.). Variable costs, such as for data media, manuals, services such as logistics service providers or packaging, are mostly low—and in the case of sales over the Internet, they are negligible. In contrast, the profit margins for consulting and maintenance services are considerably lower.

The profitability of these business models is pivotal for investors, who are often more interested in rates of return than in absolute monetary values. This often makes software providers who primarily focus on selling products and licenses more attractive for investors than software providers in the broader sense. The main reason for this is that providers of products can grow faster than companies whose sales largely consist of services. As a result, many software companies lack a consulting arm, e.g., Oracle. On the other hand, software providers who generate a large share of their sales through services have the advantage that, as noted above, they gain relatively constant revenues, which is especially significant during economic downturns, when there is little new business to be won.

It should also be noted that, the revenues from licensing and services discussed in this section are not independent of each other. Rather, the amount of revenue from licensing and providers' pricing strategies significantly influence income from services. In fact, a business model in the software industry can entail selling software at low prices—or even giving it away. This is often a rewarding and even necessary strategy, to gain an early foothold in the market or take market share from an established competitor. And in any case, it can be worthwhile to offer services in conjunction with low cost or free software, and generate profits entirely or largely from the sale of these services.

This kind of business model is especially suitable for companies that offer digital goods, such as software or music. In addition to building up revenue, there are a few other defining characteristics that differentiate software from other goods. In the second chapter, we will explore the economic background and the consequences for software markets.

Having described the basic rules of the game in [Chap. 1](#), we now turn to the economic principles in the software industry. We begin by discussing the properties of digital goods ([Sect. 2.1](#)), and go on to examine network effects on software markets ([Sect. 2.2.](#)) and the closely related issue of standardization ([Sect. 2.3](#)). We will also be looking at aspects of transaction cost ([Sect. 2.4](#)) and principal-agent theory ([Sect. 2.5](#)) that are of particular relevance to the software industry.

2.1 Properties of Digital Goods

Early in [Chap. 1](#), we introduced some of the economic properties of digital goods and software products. We will now discuss these in greater depth. A key property of digital goods is that producing the first copy generally incurs high costs, but that subsequent copies can be made at very low variable costs. Let us take a closer look at this phenomenon as it relates to the software industry. The development of software (or to be precise, the source code as the first copy), usually requires considerable investment, but there is no guarantee that a development project will be a success. For this reason, a vendor of custom software will be keen to pass on at least some of the development costs and associated risk to the customer (compare [Sect. 2.5](#)). A standard software vendor does not have this option, but bears the whole risk. At the same time, it can make a substantial profit if the product is popular. In addition, development costs are sunk costs, i.e., costs that have already been incurred and as such can no longer be influenced, so they are no more decision relevant.

However, we must point out that by assuming that variable costs are close to zero, the theory of digital goods considerably simplifies, and in many cases, oversimplifies, the real state of affairs. Only the variable costs of software licenses are close to zero. However, what are not negligible are the variable costs for providing services associated with the software, such as consulting, maintenance, and support. We will return to this issue in [Chap. 3](#).

Another important property of digital goods is that they can be copied easily without loss of quality. This explains why the costs of reproducing software are so low. Copies of digital goods are termed perfect, since the original and the copy do not differ.

The resulting problems are well known, for example, in the music business. Various peer-to-peer platforms enable users to download their own free (and often illegal) copies of music files. The same problem affects the software industry, particularly vendors whose products require little or no modification before use, and which are popular with noncommercial users. For example, office applications and computer games are often copied illegally and exchanged on file sharing networks. Several studies have attempted to estimate the loss of revenue suffered by software companies through digital piracy, including the Piracy Study published annually by the business software alliance (BSA). However, these studies have been criticized for their methodologies and assumptions. Many of them simply assume that every illegal copy leads to a direct loss of revenue, i.e., that every owner of a pirate copy would otherwise have purchased the software product. This results in immense (theoretical) losses that seem rather unrealistic, although there is no doubt that piracy costs software providers dear.

To protect their intellectual property and/or prevent illegal copying, vendors of digital goods can use digital rights management systems (Hess and Ünlü 2004). These provide protective methods based on hardware and software, including controls on access to, and use of, the software, protection of authenticity and integrity, identification via metadata, the use of copy protection, or a specific payment system.

2.2 Network Effects on Software Markets: The Winner Takes it All

In this section, we will discuss network effects, which play a significant role on software markets. The reason for this is that in addition to the functionality of a software product, especially its current and future distribution has a decided influence on its usefulness for users. This relationship is described and analyzed within the context of the theory of positive network effects. The first part of this section provides basic background information and definitions (Sect. 2.2.1). After that, we will explain why it is not always the best standards and the best software solutions that come to dominate markets with network effects, and why in many cases small software companies and startups tend to struggle (Sect. 2.2.2). First, we will describe these problems with reference to the so-called penguin effect. We will then present a model proposed by Arthur that illustrates how small chance events can determine the success of standards and software solutions in markets with network effects. We will go on to discuss why “winner takes all” so often applies to this kind of market, and how this ultimately explains why acquisitions are so frequent in the software industry (Sect. 2.2.3). Following on from this point,

we will show how software providers can exploit the existence of network effects to enhance their competitiveness (Sect. 2.2.4). Section 2.2.4 will explain both platform strategies and two-sided network effects with reference to the digital game industry. Section 2.2 concludes with a discussion of the limitations and potential extensions of network effect theory (Sect. 2.2.6).

2.2.1 Network Effects: Basics and Definitions

Michael Katz and Carl Shapiro define network effects as follows: “The utility that a given user derives from the good depends upon the number of other users who are in the same network as he or she” (Katz and Shapiro 1985, p. 424). In other words, the larger the network, the more the users benefit. Network effects are either direct or indirect.

Direct network effects arise from the fact that by employing the same software standards or common technologies, users can communicate with each other more simply and therefore more cost-effectively. The classic example of a direct network effect is the telephone: the more people own one, the more beneficial this technology becomes for users. The same principle applies to XML vocabularies such as xCBL, which are more useful the greater the number of organizations employing them. Inter-enterprise network effects also play a growing role in the context of ERP systems. The use of standard formats, for example, simplifies the exchange of business documents between different ERP systems. For this reason, stronger value-chain partners, for instance, in the automotive sector often pressurize smaller companies into deploying a compatible or identical ERP system to their own. Intercompany process standardization takes this approach a step further.

Indirect network effects, by contrast, result from the dependency between consumption of a basic good and consumption of complementary goods and services. They arise, therefore, when the wider adoption of a good generates a broader range of associated goods and services, which enhances the utility of the basic good. Indirect network effects occur in the context of standard software and complementary consulting services, operating systems with compatible application software, or with regard to the availability of programming language experts and/or tools.

Network effects lead to demand-sided economies of scale and to what is known as positive feedback respectively, increasing returns. Shapiro and Varian summarize this phenomenon as follows: “Positive feedback makes the strong get stronger and the weak get weaker” (Shapiro and Varian 1998, p. 175). Figure 2.1 shows this self-reinforcing cycle both for direct and for indirect network effects.

These network effects represent a huge motivation for users to select popular software products, and to prefer providers who can offer them strong network effects.

But there are other arguments in favor of large providers: first and foremost, they offer protection of investment, a key factor in light of the high switching costs

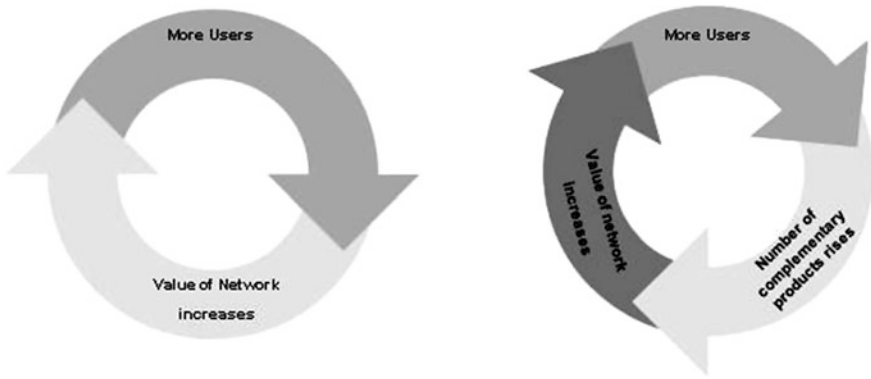


Fig. 2.1 Positive feedback loop—direct and indirect (modified from Bansler and Havn 2002, p. 819)

associated with software (see also Sect. 2.2.4). Risk reduction probably also plays a role: choosing software from a market leader such as SAP or Microsoft is unlikely to cost you your job. On the other hand, if an implementation project for open-source software ends in failure, those responsible will find it much harder to justify the decision, and failure is more likely to affect their standing within the company.

It is necessary to differentiate between the network utility and the stand-alone utility that software can deliver, i.e., the utility of the software irrespective of how many other people use it. An example of a good with both network and stand-alone utility is a spreadsheet program. The stand-alone utility lies in the functionality provided, whereas the network effect utility derives from the ability to exchange files with other users (direct network effect) or to obtain advice on how to use the software (indirect network effect). An e-mail system, on the other hand, is an example of the software that offers only a network utility, because a single user on his or her own derives no utility from the system at all.

Figure 2.2 a user's utility function. The total utility is the sum of the stand-alone and network utility. In this diagram, a positive linear correlation is assumed between the network utility and the number of users.

To measure the magnitude of the network utility in comparison to the stand-alone utility, we introduce a network effect factor Q . In addition, c represents the network utility and b the stand-alone utility of a software product. We now define the network effect factor as follows:

$$Q = \frac{c}{c + b}$$

where Q is standardized between zero and 1. The higher the value of Q , the greater the impact of network effects in comparison to the stand-alone utility. For example, the network effect factor of standard software such as Microsoft Excel

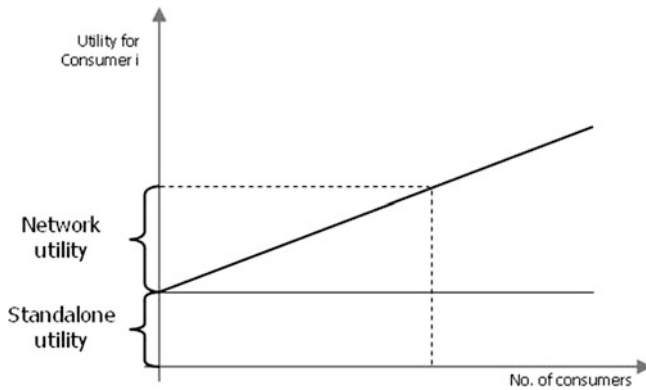


Fig. 2.2 Correlation between stand-alone and network utilities

would be somewhere between zero and 1. However, an EDI standard does not offer any stand-alone utility, so the network effect factor in this case is 1. We will return to this network effect factor in the context of assessing pricing strategies (see Sect. 3.3).

We will now examine the impact of network effects on software markets.

2.2.2 Impact of Network Effects on Software Markets

Let us begin with a straightforward example: Imagine that a newly established company has succeeded in developing a software product that offers superior functionality to Microsoft's Office package. In other words, the product offers the customer a higher stand-alone utility. Would this start-up be successful in penetrating the market? We would not like to bet on it. The problem for the small company is that the market leader—in this case, Microsoft—already offers its customers the benefit of high network effects. From a macroeconomic point of view, the optimum solution—assuming that switching costs are not too high—would be for all users to opt for the new software. This is because, as explained, if it offers a greater stand-alone utility, and if all users were to switch over; corresponding network effects would arise over time. Nevertheless, a wholesale changeover is unlikely to happen, because the existence of network effects can lead to a lock-in, a technically inferior technology (cf. David 1985; Liebowitz and Margolis 1994). As a result, a company that has an installed base in a network market has such a significant competitive lead that is extremely difficult for other players to catch up. In our example, Microsoft has a competitive lead.

2.2.2.1 The Penguin Effect

According to Farrell and Saloner, the fact that the installed base often hinders the changeover to a technically superior standard is due to informational and resulting

coordination problems (Farrell and Saloner 1985). Their reasoning goes like this: The sum of the utilities enjoyed by all market players could be increased, if these opt to transition to a new (technically superior) standard—in our example, if they choose the startup’s software. However, the users are unsure whether this transition is actually taking place. A potential switcher with incomplete information is facing the problem that the other market players may not follow suit, and the greater stand-alone utility enjoyed when using the new standard cannot compensate for the network utilities forfeited. The uncertainty about how the other market players will respond can encourage a company to maintain the status quo. This coordination problem is also termed the penguin effect, based on the following analogy: hungry penguins are standing at the edge of an ice floe. Out of fear of predatory fish, they hope that other penguins will jump into the water first, to check out the risk of falling victim to a predator. As soon as some of the birds have taken the plunge, the danger for the others is reduced, and the free-rider penguins follow suit (Farrell and Saloner 1987).

In light of this effect, software providers face a startup problem. Although every new player in a market has difficulties to contend with, they are more significant for companies in markets with network effects. The startup company in this scenario not only has to effectively promote the product itself, but also assure potential buyers that it is going to prevail in the marketplace and generate network effects. This startup problem caused by the penguin effect is also referred to as “excess inertia”. The converse problem, i.e., “excess momentum”, or an excess of different standards, can also arise.

The penguin effect is one of the main reasons why an inferior standard often manages to gain the upper hand; there are many examples of this phenomenon:

For a long time, the VHS video standard dominated the market, although Betamax was technically superior.

OSI protocols did not succeed in becoming standard at all levels, despite their high quality in technical terms. Today, Internet protocols dominate the market.

The QWERTY keyboard layout is said to be less efficient than alternatives, such as the later Dvorak layout, which conforms with ergonomic principles. Converting to a potentially more efficient layout is prevented by both direct and indirect network effects.

What we learn from these examples is that it is very difficult to replace a standard, once it has become firmly established on the market.

2.2.2.2 Diffusion Processes: The Model Proposed by Arthur

According to Arthur, small chance events can determine which standard ultimately prevails (Arthur 1989). The term “path dependence” describes the effects of earlier, sometimes random events that occurred during the diffusion process on the market structure (David 1985, p. 322). The model described below clearly illustrates how competition between two technologies plays out when both of them generate network effects.

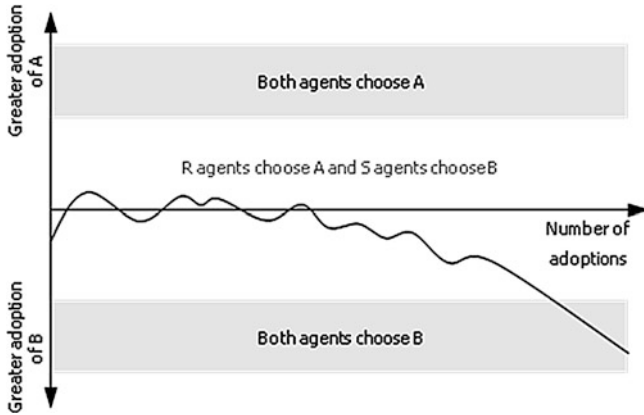


Fig. 2.3 Path dependence due to increasing returns (Arthur 1989, p. 120)

The model takes two technologies, A and B, as a starting point. In addition, there are two types of users, which Arthur terms R and S agents. The total utility of a technology is then the sum of the stand-alone and network utilities. The model uses the following parameters to denote the stand-alone utility:

- a_r stand-alone utility of technology A for R agents
- a_s stand-alone utility of technology A for S agents
- b_r stand-alone utility of technology B for R agents
- b_s stand-alone utility of technology B for S agents

We shall assume the following: technology A has a higher stand-alone utility for R agents than technology B ($a_r > b_r$). Conversely, technology B has a greater stand-alone utility for S agents than technology A ($a_s < b_s$).

As shown in Fig. 2.2, we assume that network utility increases linear with the number of users. n_a denotes the number of users of technology A and n_b the number of users of technology B.

Arthur’s model shows how, based on the agents’ decisions, a technology comes to dominate the market. To this end, Arthur develops a simulation model in which an R or an S agent opts for one of the two technologies at random. The agents choose the technology that offers them the greatest total utility. The total utility of technology A for the R agent is $(a_r + r \dots n_a)$, while the corresponding total utility of technology B is $(b_r + r \dots n_b)$. The parameter r stands for the marginal utility generated by each additional user. It is assumed that the S agents take their decisions on the same basis. Figure 2.3 shows the results of applying this simulation model.

In the white area shown in the diagram, R agents choose technology A and S agents opt for technology B, i.e., each chooses the technology that gives them the greatest stand-alone utility. However, if many S agents choose technology B, the network utility of technology B increases. From a given number of users upward, this prompts R agents to opt for technology B, too, as the higher network utility of

this technology more than compensates for the lower stand-alone utility for the R agents.

This simple model shows two things clearly: first, so-called early adopters play a decisive role in the struggle for market share. Second, that it is often chance events that determine how a given technology fares in markets with network effects.

2.2.3 Structure of Software Markets

For a long time, economists have recognized that network effects often lead to monopolies. For this reason, markets displaying strong positive network effects and feedback loops are frequently termed “tippy”: “When two or more firms compete for a market where there is strong positive feedback, only one may emerge as a winner. It’s unlikely that all will survive.” (Shapiro and Varian 1998, p. 176). Multiple standards or technologies seldom exist side by side. A dominant technology crowds out the others (Besen and Farrell 1994, p. 118). Therefore, they are known as winner-takes-all markets.

An observation of today’s software markets bears out these theoretical considerations. Looking at markets for standard software solutions, for example, it is obvious that the number of providers has fallen sharply. A few years ago, the market still had room for alternatives to Microsoft products for browsers or office solutions, such as Netscape Navigator, the WordPerfect word processing system, or the 1-2-3 spreadsheet application. Now, there is really only one powerful competitor to Microsoft on these markets: the open source community.

The browser wars

The browser war between Microsoft and Netscape started in 1995. Netscape rose to prominence with its Netscape Navigator web browser. Prior to 1995, it had a market share of more than 80 %. The battle between these two companies commenced when Microsoft finally recognized the growing importance of the Internet, and developed its competing product, internet explorer (IE). During these wars, there were two issues of particular interest that will be described in more detail below: first, the importance of open-source products to rival commercial software, and second, the effect of price bundling on a business.

Phase I: The beginnings of the WWW

One of the leading web browsers to emerge in the early 1990s, when the Internet was really taking off, was Mosaic. It was the only browser to boast a graphical user interface that enabled users to surf the web. Established in late 1993, Netscape launched the first version of its Netscape Navigator browser in 1994. By 1995, Netscape had built up a monopoly on the browser market.

Phase II: 1995–1998

When Microsoft decided to enter the Internet market, the first version of the Internet Explorer was developed. The company had two decisive advantages over Netscape: first, Microsoft had far more financial resources at its disposal to develop browser software. Second, Microsoft was able to create product packages, and started to include Internet Explorer in its software offerings. From Windows 98, Internet Explorer was a fixed component of the operating system (along the lines of: “If it’s installed, it will be used”). Since Windows is preinstalled on about 95 % of all new PCs, Microsoft’s price bundling strategy soon paid off. In the following period, Internet Explorer’s market share rose from below 3 % initially, to over 95 %. In 1998, with its market share below 4 %, Netscape was finally compelled to admit defeat. Netscape then published the source code of its Navigator and converted it to the Mozilla open-source project.

Phase III: 2004 to the present

This phase is also referred to as the second round of the browser wars. In 2004, more and more security gaps in Internet Explorer were coming to light enabling Mozilla to regain market share. The Mozilla Organization launched Firefox 1.0 (a slimmed-down browser) in November 2004. It spread rapidly thereafter, not only because of Internet Explorer’s security deficiencies. Firefox offers a range of practical functions (tabbed browsing, find-as-you-type, Download Manager, etc.), which were contributed by open-source developers. Microsoft failed to upgrade the outdated technology underlying Internet Explorer 6.0 fast enough to prevent the modern, open-source browser, Firefox, from capturing significant market share. It was not until version 7 (many of whose “new” functions were copied from Firefox) that Internet Explorer regained a positive image on the market.

A monopolistic market structure such as that for office software often poses problems for users. However, Microsoft did not come under criticism for imposing a (higher) so-called Cournot price—as predicted by traditional microeconomics. Rather, it was for violating competition law that the European Commission levied a large number of fines on Microsoft, and instructed the company to disclose interface specifications and debundle certain of its products or face further penalties. Both demands benefit the other software providers by making it easier for them to develop and market products that are compatible with Microsoft’s standards.

2.2.4 Network Effects as a Competitive Factor

Our discussions so far indicate that network effects can represent a decisive competitive advantage for a provider. Once a software solution has become widely

adopted, a lock-in effect will occur. As a result, potential rivals are faced with a start-up problem whose magnitude correlates with the network effect factor.

At the same time, an analysis of the competitive environment must take into account that a user who changes to a different software solution generally incurs high switching costs. This applies particularly in relation to ERP systems, which explains why they are seldom replaced in practice. But what accounts for these high switching costs? One of the principles behind the ERP market is that this type of software models and shapes the user's business processes. Changing to a different provider would therefore result in considerable costs in terms of organizational changes. In addition, especially for standard software with a high network effect factor, there is uncertainty about how many other users will change over to a new software standard—compare the penguin effect discussed in [Sect. 2.2.2.1](#). The switching costs will also increase, the longer an ERP software product has been in use, because over time, it will become better integrated into the overall environment. Extensive system customizations increase the switching costs still further.

How can a software provider motivate users to switch over despite the lock-in effect? One possibility is to subsidize the change. Moreover, the vendors could pursue a low-price-strategy (see [Sect. 3.3](#)).

Major providers who are already able to offer their customers network effects may find it lucrative to maintain their competitive edge by not disclosing interface specifications in full. This makes it difficult for other providers to offer compatible products. In today's world, however, putting limits on compatibility is unlikely to be popular with customers. Users have long since realized that open standards make them more vendor-independent and above all, can simplify IT integration both within and between companies.

However, the product compatibility question pans out very differently for small software companies and for providers whose solutions have yet to be launched. These providers have no choice but to either go for completely open standards or develop products that are compatible with those of the major players. A very good example is provided by Business Objects, a provider of business intelligence software:

Business Objects was established in 1990 by two Oracle employees. The company began with an application that enabled Oracle customers to create database queries intuitively, without any knowledge of SQL. In the early years, the company targeted only Oracle customers and their application only supported Oracle databases. In other words, they pursued a niche strategy and did not include other databases at first. By concentrating on Oracle customers, Business Objects was able to forge a close partnership with Oracle. Oracle saw Business Objects products as complementary to its own; they were not rival offerings, but helped drive sales of its own database technology. Later, Business Objects built other partnerships, such as with IBM, and became a well-known provider of business intelligence technology. In 2007, the company was acquired by SAP.

The Oracle database system first served as a springboard for the products and services offered by Business Objects. This model can be found in many other areas of the software industry, such as in the games sector, which we will look at below.

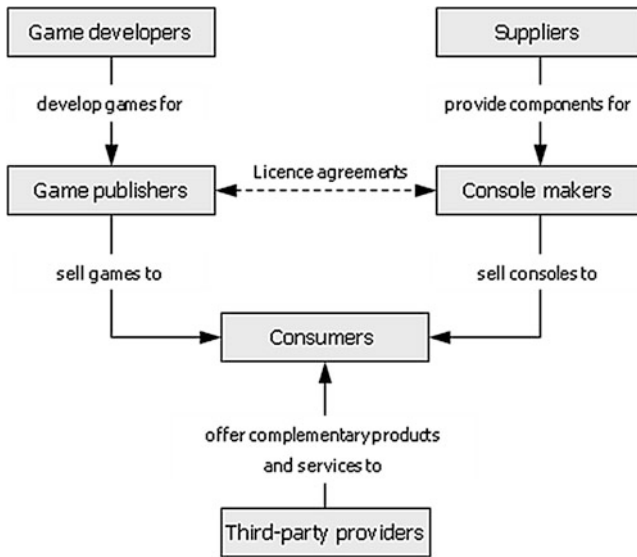


Fig. 2.4 Structure of the value chain in the digital games industry

2.2.5 A Case Study: Two-Sided Network Effects and Platform Strategies in the Digital Games Industry

In this section, we will describe a branch of industry positioned somewhere between the software and the media industry: digital games. We will use this market to explain the impact of two-sided network effects and the importance of platform strategies. Our focus will be on console games, because the varied results of network effects are particularly apparent in this market.

2.2.5.1 Overview of the Digital Games Industry

Digital games have grown into a sizeable market for the entertainment industry. The USA plays a leading role, not only due to the size of its retail market, but also because most of this sector’s software providers are based there.

We will start by looking at the players and the value chain in the digital games industry. The main players are console manufacturers, suppliers, game developers, game publishers, consumers, and third-party providers. The value chain in this sector is shown in Fig. 2.4.

Console manufacturers develop and sell the consoles needed to play the digital games. For this reason, we refer to this hardware below as platforms. At present, the main console vendors are Microsoft, Nintendo, and Sony. They are assisted by *suppliers* who provide certain console components. For example, specialized sound and graphics chips and CPUs are produced by manufacturers like NVIDIA, ATI, IBM, and Intel.

Game publishers, such as Electronic Arts and Activision, reproduce and market games for these consoles. They act as intermediaries between the game developers and the console makers.

The games are developed by in-house or independent *game developers* or by studios such as Pyro Studios, Rockstar North, and Deck13. Either the developers are commissioned and funded by the game publisher to create a particular game, or they themselves approach publishers with a prototype. A unique feature of this market is that game publishers require a license from the console maker before they are allowed to market a game for that manufacturer's console. Under this model, the console manufacturer receives a portion of the publisher's sales revenue and thus has a stake in a game's success.

Third-party providers (e.g. manufacturers of peripheral equipment, magazine publishers, or video rental stores) offer complementary products and services such as gaming magazines, or joysticks, and memory cards. *Consumers* purchase consoles, games, and other complementary products and services via corresponding distribution channels (department stores or online retailers).

2.2.5.2 Two-sided Network Effects on Digital Games Markets

As mentioned above, the consoles represent the (industry) platform on which the games can be played. This constellation is comparable with other areas of the software industry. For example, operating systems underpin application software, while integration platforms such as SAP's NetWeaver and IBM's WebSphere form the basis for service-based software. As a rule, the platform providers do not have the resources to develop the applications, services, or games themselves (Cusumano 2004, p. 75). That is why they often work closely with other software and service providers. And very often, the winner-takes-all principle described earlier applies to platform/console providers, too. This has led to a consolidation on this market. In the early days, there were several console providers, whereas today there are only three: Microsoft, Nintendo, and Sony. The fierce competition on this market is often referred to as the "console wars". The manufacturers generally launch a new generation of hardware about every 5 years. Figure 2.5 shows the leading consoles on the European market. Some of them—such as Atari's Jaguar or Sega's Dreamcast—could not compete for very long, despite their technical superiority. Others—such as the Nintendo NES and the Atari 2600—held their own for years, despite the appearance of new, more advanced consoles. The network effect theory introduced earlier can explain why technically inferior systems can dominate markets.

The success of a console or platform does not depend on its technical merits or price alone, but also on the game software available for it. Therefore, this is a market with indirect network effects as the utility and popularity of a console is primarily determined by the availability of attractive games, which are complementary goods (compare Sect. 2.2.1).

The structure of the value chain in the digital games industry can also help us illustrate network effects of another kind, termed *two-sided network effects*.

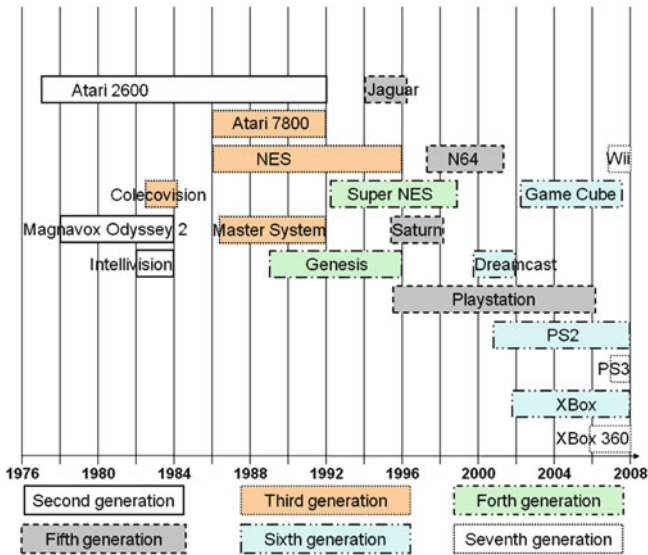


Fig. 2.5 Console generations in Europe (from Wikipedia, reworked)

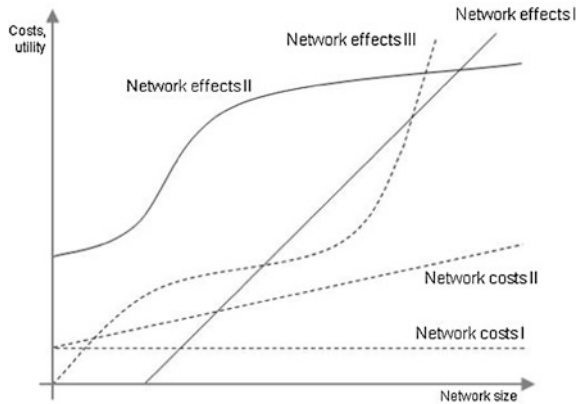
Multiple groups form around a given platform, whereby the utility that the platform offers one group depends on the number of members in the other group. Armstrong gives an illuminating example: “For instance, a heterosexual dating agency or nightclub can only do well if it succeeds in attracting business from both men and women” (Armstrong 2006, p. 1). On the digital games market, the two groups comprise game publishers and consumers. The game publishers benefit when large numbers of consumers own a console, resulting in high demand for the games played on it. On the other side of the coin, consumers benefit when many game publishers develop games for a console, because that increases the supply of games, which is the console’s indirect network utility.

In this section, we have restricted our discussion to the console games subsector, in order to illustrate how two-sided network effects work. There are, however, other sectors in the digital game industry, such as mobile games, browser games, and massive multiplayer online games, all of which are predicted to soar in popularity.

In recent years, we can observe the emergence of an increasing number of apps and games which are offered for social media platforms like Facebook. Market leader for the game industry in this sector is Zynga with titles like Farmville. Zynga is linked very closely to Facebook and generated approximately 12 % of Facebook’s revenue in 2011.

In the following section, we will describe the limitations of network effect theory, and possible ways of extending the theory.

Fig. 2.6 Possible cost and utility functions depending on network size (Weitzel 2004, p. 30)



2.2.6 Limitations of Network Effect Theory

Although network effect theory has given rise to some interesting insights with respect to explaining the widespread adoption of standards, there are limits to its applicability. In light of this fact, the following extensions, among others, have been suggested (Weitzel et al. 2000).

2.2.6.1 Homogeneous Network Effects and Costs of Network Size

An important potential extension of traditional network effect theory models relates to assumptions about the dependence of the network effect utility on the number of users. Often, as in the preceding sections, a linear utility function is assumed, i.e., every additional user leads to a constant utility increase for the users as a whole. However, degressive or progressive functions are equally conceivable. With the former, the n th user produces a smaller utility than the $(n-1)$ th user. With the latter type of function, the n th user produces a greater utility than the $(n-1)$ th. Similarly, various models also make differing assumptions about the cost functions in relation to the network size. Alternative functions are shown in Fig. 2.6.

However, all the utility functions shown in the figure—whatever form they take—ignore the individual nature of communications relationships. We believe that individualization is essential to decisions about employing standards, such as EDI. It is obviously extremely relevant to an automaker which standards its suppliers use. On the other hand, the company will be relatively unconcerned about the standards used by enterprises with which it has no business relationships at present and is unlikely to in future.

2.2.6.2 Type of Network Effect Utility

In the models proposed to date, the network effect utility is predominantly treated in the abstract and not specified. However, this means that the utility is not concretized, either. This raises the question as to what concrete utility, a given network effects actually offers.

2.2.6.3 Normative Implications

Most of the literature treats standardization decisions from a macroeconomic perspective, particularly with regard to the consequences of the existence of network effects in terms of market efficiency. The intention of most articles is to explain how users make standardization decisions. However, they do not, as a rule, offer any concrete recommendations.

The standardization problem touches on the limitations of network effect theory discussed above. First, a detailed, actor-specific method is proposed for modeling the costs and utility of network effect goods; second, the network effect utility is concretized; and third, recommendations will be offered to users on the extent to which standardization is the best solution.

2.3 The Standardization Problem

2.3.1 Approach and Background

The application systems used in many companies and groups have grown up over long periods, often in an uncoordinated manner. This gives rise to heterogeneous IT environments which, due to incompatibilities, hinder the flow of information between different parts of the organization. According to an oft-cited rule of thumb, developing and maintaining interfaces between incompatible subsystems can account for up to half of the entire IT budget. The use of standards is a key method of reducing these integration costs. Examples include the use of EDI standards to exchange information between companies, and SOA platforms deployed to enable the seamless integration of services from different providers on the basis of Web service standards.

An alternative to standards is the use of converters. However, this generally leads to higher costs than standards, because integrating n functions or business units can require up to $n * (n - 1)$ converters (Wüstner 2005). Furthermore, the consistent use of converter-based solutions usually reduces the flexibility of the overall IT environment. For these reasons, we will not discuss this option in any more detail.

The model introduced below looks into user decisions on the use of standards to link applications. However, providers can derive strategy recommendations from the model by employing it to anticipate the decisions, users are likely to make (Buxmann 2002). Like the model proposed by Arthur, we initially assume that a software standard offers a user both a stand-alone utility and a network effect utility. The stand-alone utility could be the functionality of an ERP system, for example. In principle, standard investment evaluation methods—such as multi-attribute utility analysis or the net present value method—can be used to compare the stand-alone utility of multiple software solutions. How can we concretize the network effect utility, which is largely treated in the abstract in network effect theory? In the following discussion, we will assume that using shared software

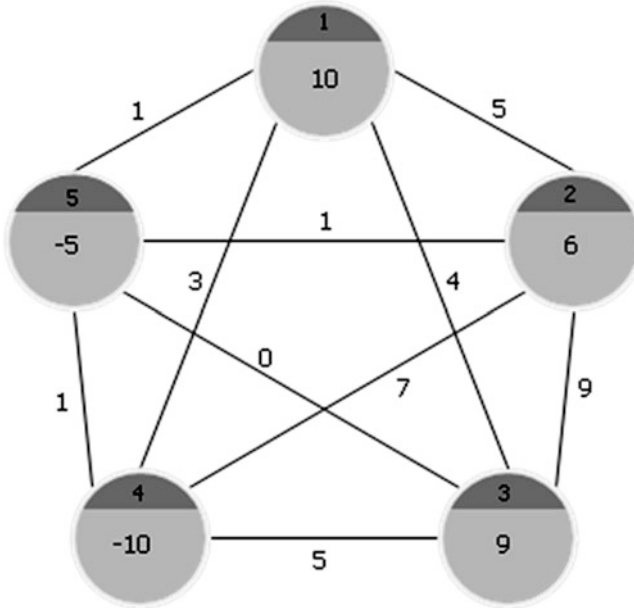


Fig. 2.7 Example illustrating the standardization problem

standards leads to savings in information costs. This term encompasses all costs incurred for exchanging information. For example, using a common EDI standard enables savings on human resources, postage, paper, etc. The common use of the same Microsoft Office solution generally saves money, time, and trouble when exchanging documents. Similarly, information cost savings can be achieved when a standard software product supports business processes across multiple departments.

On the other hand, costs are incurred for the procurement and implementation of software standards. These will be termed standardization costs in the following sections.

We will now simplify our reflections by taking a static approach. That means, for instance, that the cost of introducing a software standard will be assigned a specific value in our model. This is of course a simplification, because the costs associated with an investment of this kind differ from year to year. For example, users have to bear one-time licensing costs and also annual maintenance and service costs. This simplification is, nevertheless, unproblematic for two reasons: first, we can interpret the value assigned to the standardization costs as a discounted value of the payments incurred over the years; second, a dynamic approach that explicitly takes into account different points in time does not change the main model predictions that will be derived below.

The standardization problem is formulated on the basis of a graph as shown in Fig. 2.7. The actors are represented as nodes and the relationships between them as edges. The model is formulated in a general way: in other words, the nodes can stand for companies, e.g. in a supply chain, or organizational units within an enterprise.

The stand-alone utility and the standardization costs are modeled nodes related. The information costs that can be saved are edges related. If, to take the simplest case, we assume that there is only one standard available, meaning that each actor has to take a yes/no decision for or against introducing a given standard, we can label the stand-alone utility of node i as b_i and the standardization costs of node i as a_i ($i = 1, 2, \dots, n$). If the decision about introducing a software standard were to be taken with reference to the nodes and in isolation, only the difference between the stand-alone utility and the standardization costs would have to be determined for each node. If this net stand-alone utility is greater than zero, the software standard is a worthwhile investment, otherwise it is not.

But such an isolated treatment is not useful in practice, because the IT integration costs in particular are usually very high. In our model, the information costs along the edge from node i to node j are termed c_{ij} . These can be reduced if both actors implement the common standard, such as the same EDI standard, the same standard software or the same communications protocol. Of course, it is unrealistic to assume that the use of common standards will reduce the information costs to zero. But this does not pose a problem for the model, because the edge values may also be interpreted as the difference between the information costs without and with the common software standards.

Figure 2.7 shows a simple example. The respective net stand-alone utility is shown in the nodes, while the edges represent the communication costs that can be reduced. Using this example, we can demonstrate that, unlike classical network effect theory, this approach makes it possible to model the communications relationships between certain actors on an individual basis. That means the substantial demand for communications between nodes 2 and 3 is shown by edge costs of 9. These can be saved by adopting a common standard. In contrast, the need for communications between node 5 and the others is slight, and so using the common standard can only lead to minor savings in information costs.

In the first analysis, as shown above, the ideal case is when the standard is adopted by all of the nodes for which the net stand-alone utility is positive. In this case, this applies to nodes 1, 2, and 3. But standardization can be advantageous even for a node with a negative net stand-alone utility. This is precisely the case when this node has close communications relationships with others, and in consequence, substantial information cost savings can be achieved. In our example, this applies to node 4. Although the net stand-alone utility for this node is negative, standardization is worthwhile overall, as information costs amounting in 15 can be eliminated if nodes 1, 2, and 3 also introduce the standard. On the other hand, standardization makes little sense for node 5. Here, the information cost savings of 3 are lower than the negative net stand-alone utility of 5.

2.3.2 The Central Standardization Problem as An Optimization Problem

In formal terms, we can represent the standardization problem as an optimization calculation as follows:

$$\max F(x) = \sum_{i=1}^n (b_i - a_i)x_i - \sum_{i=1}^n \sum_{\substack{j=1 \\ j \neq i}}^n c_{ij}(1 - x_i x_j)$$

s.t.

$$x_i \in \{0, 1\}$$

Decision variable x_i takes the value of 1, if actor i employs the standard. Only in this case can the stand-alone utility b_i be achieved, and only then will standardization costs a_i incur. The information costs c_{ij} can only be saved if both actor i and actor j employ the standard. The goal function also shows that the model tends toward optimization of the entire graph. In one application, for example, a central entity, e.g. the Chief Information Officer and his staff, could look for the optimal solution for the entire company. This, however, can mean that individual actors are worse off than before.

The above model only encompasses situations in which there is exactly one standard. In contrast, the extended standardization problem also looks at situations in which the decision maker can choose between several standards—we will not show the mathematical details of this model in this book (Domschke and Wagner 2005).

To illustrate the model, perform optimization and simulations, we have developed a prototype. Figure 2.8 shows the optimum standardization solution from a central viewpoint for a complete communications network with seven actors and a choice of three standards. In this example, the optimum solution is to implement multiple standards in the network, and even to provide individual actors with more than one standard.

The prototype makes it possible to generate communication networks with widely different structures and examine them under diverse cost distributions. For example, it is possible to investigate the effects of the network topology on the advantages of specific standardization options, such as partial standardization or full standardization. Figure 2.9 gives an example of this kind.

We have explored this model both by analytical means and via simulations (Buxmann and Schade 2007). This has permitted us to derive the following three basic propositions:

1. The best solution is often to standardize the network fully or not at all.
2. The larger the network, the more worthwhile full standardization will be.
3. If multiple software standards are available, it is seldom the best option to employ several of them.

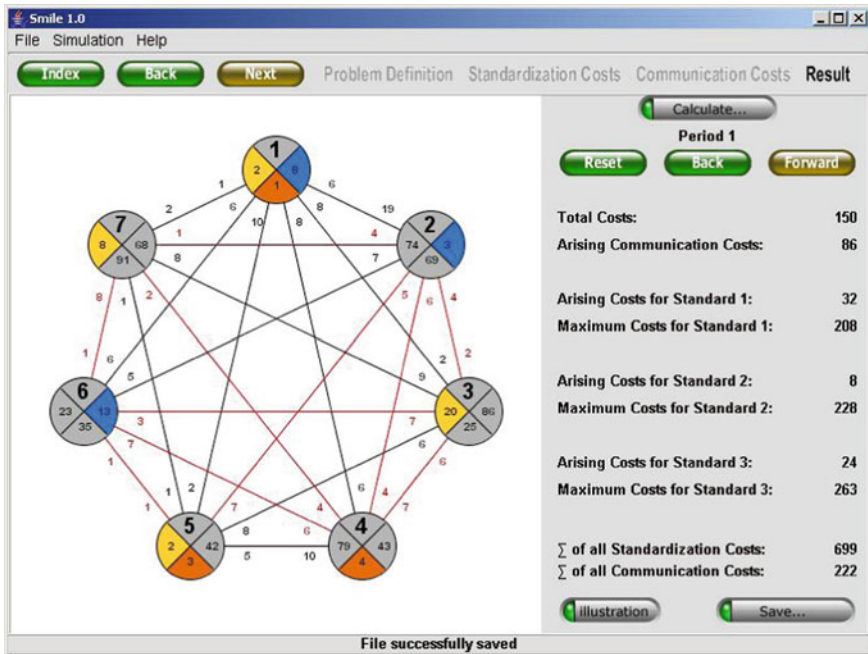


Fig. 2.8 Solution to the standardization problem for seven actors (Schade and Buxmann 2005)

That also means that best-of-breed solutions generally do not represent an ideal solution for the user organization.

This means that the best option for a major software provider is generally to offer solutions for all areas (nodes) of a network. A standard solution is usually beneficial to the users. This is especially true if the customer organization is large—expressed, in terms of the model, as the number of nodes n . One exception to this rule (which is, however, relatively rare in practice), is a scenario in which one area of the business (one node) has few, if any, information relationships with other areas (nodes). In this case, the customer’s best option may be to employ multiple software standards in accordance with a best-of-breed strategy.

Obviously, smaller software providers and niche providers must ensure their products comply with compatible standards and are therefore, attractive to users who wish to save on information costs (for integration or conversion).

2.3.3 The Decentralized Standardization Problem: A Game Theoretical Approach

Our considerations so far have been based on the assumption that a central entity exists that decides about adopting standards for all nodes. This would mean that a CIO and his team are able to choose (and enforce) the respective software standard

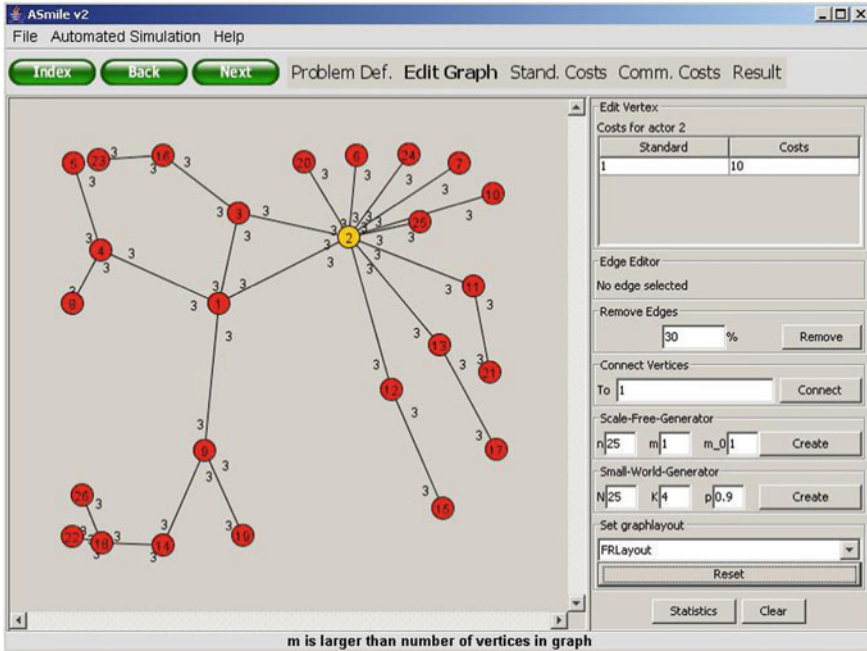


Fig. 2.9 Examination of various network topologies

for all organizational units. From an interorganizational perspective, it is reasonable to assume that the strongest player in a supply chain can impose the software standard on its partners. But what happens when every node takes standardization decisions independently (Buxmann et al. 1999; Weitzel 2004; Heinrich et al. 2006)?

To describe this decentralized problem, we will take an approach based on game theory (Weitzel 2004). We will simplify the problem, by assuming a single-standard problem and examining just two actors i and j . This is in fact perfectly sufficient to illustrate the differences between a centralized and a decentralized decision and the resulting consequences. Each of these actors takes a decision to implement or not to implement the given standard. We now introduce directed edges between the actors with the help of which we can show that adopting a software standard can benefit one participant more than another. In formal terms: if both actors adopt the given standard, actor i can save information costs c_{ij} , while actor j can save c_{ji} . This general problem can be modeled on the basis of non-cooperative game theory, as shown in Fig. 2.10:

This matrix may be interpreted as follows: the squares give the results of the four possible strategy combinations based on the two alternatives (standardize or do not standardize) facing actors i and j , respectively. Starting at the top left, if both actors adopt the software standard, both can save their information costs. What remains is the net stand-alone utility for actors i and j , which we will term u_i and u_j , respectively. In the top right-hand square, only actor i is adopting the

		Actor <i>j</i>	
		standardizes	does not standardize
Actor <i>i</i>	standardizes	u_i / u_j	$u_i - c_{ij} / -c_{ji}$
	does not standardize	$-c_{ij} / u_i - c_{ji}$	$-c_{ij} / -c_{ji}$

Fig. 2.10 The standardization problem in the light of game theory

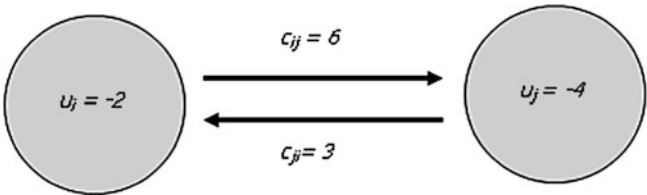



Fig. 2.11 An example scenario for decentralized standardization decisions

standard. As a result, actor *i* achieves $u_i - c_{ij}$ —the net stand-alone utility for introducing the software standard minus the information costs that actor *i* cannot save because actor *j* has decided against adopting the standard. In this scenario, actor *j* makes a loss to the amount of information costs c_{ji} . At the bottom left, the same applies, but the actors’ roles are reversed. The bottom right square describes the case where neither of the actors adopts the standard. As a result, there are no standardization costs, and both actors have to bear the information costs c_{ij} and c_{ji} , respectively.

In game theory, a situation where no actor has an incentive to change his decision (in this case in favor of or against adopting the software standard) is known, in simple terms, as a Nash equilibrium. The Nash equilibrium is therefore a combination of strategies that represents a stable state. This game-theory-based matrix can now be examined with different parameter constellations to discover what equilibria form. For our purposes, a particularly interesting and instructive scenario is one where, from a global perspective, it would appear ideal for both actors to adopt the standard, but where, from a decentralized viewpoint, only one of the actors benefits from adopting the standard. Let us look at Fig. 2.11.

It is obvious at first glance that from a global perspective, the ideal situation would be for both actors to standardize. Although the net stand-alone utility is negative, i.e., the standardization costs are higher than the gross stand-alone utility, standardization is still worthwhile because information costs totaling 9 MUs (monetary units) can be saved. The negative net stand-alone utility, by contrast, only amounts to -6 MUs. So the total saving is 3 MUs. The problem now is that there is no central entity that takes the decision for both actors and seeks the optimum

		Actor <i>j</i>	
		standardizes	does not standardize
Actor <i>i</i>	standardizes	-2 / -4	-8 / -3
	does not standardize	-6 / -7	-6 / -3



 Nash equilibrium

Fig. 2.12 Modeling the example using game theory

solution from a global viewpoint. However, assuming a decentralized perspective for both actors, it is clear that only actor *i* benefits from both sides adopting the software standard: the information cost savings of 6 MUs are greater than the negative net stand-alone utility. The picture is different for actor *j*, for whom adopting the standard does not make sense because *j*'s negative stand-alone utility is higher than the information costs that can be saved by adopting the standard.

Figure 2.12 shows the results matrix derived from game theory for this example.

This matrix is interesting in that it shows that the standardization solution (top left square) cannot be seen as a Nash equilibrium, although this is the ideal solution from a global perspective. Obviously, actor *j* has an incentive not to choose this solution and save 1 MU.

2.3.4 The Standardization Problem: Lessons Learned

A comparison of the decentralized with the centralized model highlights the fact that decentralized decisions about adopting software standards tend to lead to a lower—and from a global perspective, frequently suboptimal—degree of standardization than centralized decisions. One reason for the actors' reluctance to adopt standards when faced with a decentralized standardization problem may be found in the penguin effect discussed in Sect. 2.2.2.1. A second is that different actors benefit to different extents when all of them adopt a standard.

In order to achieve an ideal result from a global perspective with decentralized decision structures, financial incentives could be worthwhile for those actors who lose out when the standard is implemented (Heinrich et al. 2006). For example, in the example given in Fig. 2.11, actor *i* could make a compensatory payment to actor *j* to encourage *j* to follow suit. This is problematic for two reasons: first, determining the appropriate size of the payment. Second, it will be all but impossible to negotiate compensation payments with all those many business partners and determine a distribution acceptable to all actors (see also Sect. 3.1.1 Co-operations

in the software industry). In practice, the stronger players tend more or less to dictate the standard to the weaker ones—it was during negotiations on EDI standards that the phrase “EDI or die” was coined. However, a recent empirical study showed that standards are often introduced by business partners cooperatively. The study found that in about 70 % of cases in the automotive industry, the major carmakers assisted their smaller partners, particularly systems suppliers, with the introduction of software solutions (Buxmann et al. 2005).

Furthermore, the standardization problem reveals that it is generally preferable for user organizations to implement just one standard. In consequence, internal network effects (e.g. within an enterprise or a supply chain) reinforce the position of major software providers. Attempting to introduce a large array of different systems, especially in large organizations, will often lead to high information costs, e.g., in the form of integration and conversion costs. In many cases, therefore, the best solution from a global perspective is to use an integrated solution from a single vendor—complemented, perhaps, with compatible products from smaller software providers and niche providers. Best-of-breed solutions are only advantageous for the user when there is very little exchange of information and so the information costs between different areas of the business are low—a rare scenario in the real world. A key issue for users and providers is to what extent service-oriented architectures and platforms can contribute to reducing communication costs through the use of open standards, and therefore make best-of-breed solutions more attractive (we will discuss service-oriented architectures in more detail in [Sect. 4.7](#).)

From a provider’s perspective, too, it is seldom worthwhile to incorporate a host of different standards into a solution (Plattner 2007, p. 3). It makes more sense to select the right standard in terms of giving customers interoperability between different vendors’ platforms. Against this background, it is, of course, vital for providers to participate in the development of standards in order to assert their strategic interests. However, during the development of standards, problems frequently arise for the following reasons (Plattner 2007, p. 3):

In standardization processes, providers often represent different economic interests (we have looked into this issue in [Sect. 2.2.4](#))

Standardization processes are often slowed down by the typical features of committee work (Plattner talks about “committee thinking”).

For employees, participation in standardization bodies is often merely an extra task in addition to their main jobs, which is reflected in the way they prioritize their activities.

2.4 Transaction Cost Theory: In Search of the Software Firm’s Boundaries

The position a company assumes within the value chain is of great importance. In essence, this is a classic make-or-buy decision: what should the company produce or develop in-house, and what should it source from third parties? Transaction cost

theory provides some interesting insights into these questions. In [Sect. 2.4.1](#), we introduce the starting point and elements of transaction cost theory. In [Sect. 2.4.2](#), we present a general model for economically efficient division of labor among companies. In addition, we demonstrate where transaction cost theory can be applied within software companies. [Section 2.4.3](#) deals with the impact on transaction costs of structural changes—such as those brought about by the emergence of new communications technologies. [Section 2.4.4](#) considers the way intermediaries can lower transaction costs.

2.4.1 Starting Point and Elements of Transaction Cost Theory

In markets characterized by division of labor, players are linked by many and varied exchange relationships. This is the starting point for transaction cost theory (Williamson 1985). But the theory focuses not on the exchange of goods and services per se, but on the transfer of property rights that precedes the exchange. This transfer is referred to as transaction. For example, a provider handing over a piece of tailor-made software to its customer is a transaction. Further examples include integrating a module developed by an offshore service provider, or forwarding functional specifications from department A to department B within a company. To avoid confusion, we would like to point out that the term “transaction” is also used by the software industry in the context of database systems. This, however, is completely unrelated to the way the term is used in economics.

Transactions cause costs—an insight, incidentally, that economists ignored for many years. In particular, costs are incurred while gathering information and communicating with the transaction partner during deal initiation, agreement, completion of the transaction, and for the purposes of control/monitoring and fine-tuning. Let us take a look at an example. To draw attention to its offering, a custom software vendor must participate in trade shows and conferences and make regular appearances in the trade press. The client will also want to gain an overview of the market. This means that both companies will incur significant costs associated with initiating the transaction (setup costs). Further costs will be incurred for negotiations as to the scope of service and the price of the software to be developed. Other costs include those for policing and enforcement (for example via testing) and potentially for later modifications (e.g. if requirements should change). All in all, the provision of custom software is a transaction cost-intensive process.

This is where transaction cost theory comes in. It is concerned with determining what organizational form incurs the lowest transaction costs in a given situation. The approach can be applied both to the division of labor between enterprises and among departments within a company. We will look at the division of labor among market players in greater detail below. But first, let us examine the major determinants of transaction costs, and briefly tease out the theory’s underlying assumptions.

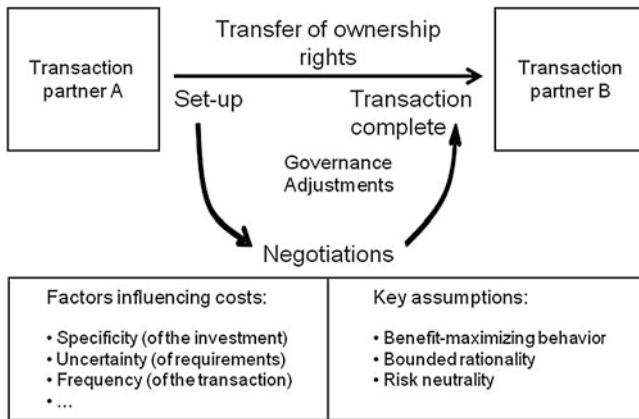


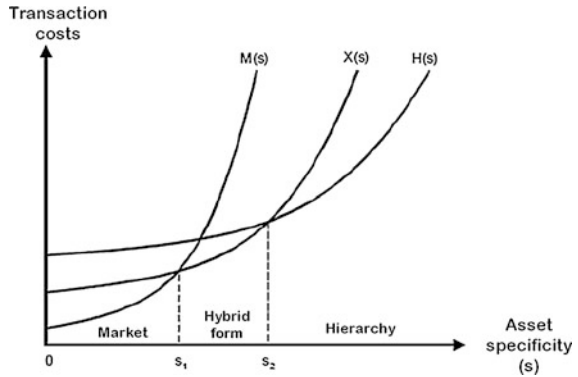
Fig. 2.13 Basic model underlying transaction cost theory

In transaction cost theory, the level of investment specific to the transaction (often termed simply “specificity”) is regarded as the main determinant of the cost of a transaction. The reasoning is quite straightforward. While the purpose of transaction-specific investments is to cut production costs, high transaction-specific investments create considerable dependency between the parties. Because once an investment has been made, the company cannot switch transaction partners without incurring losses, and it must be prepared to renegotiate terms with the existing partner. For example, large-scale outsourcing of IT services can create a dependency of this type. So clearly, transaction-specific investments will initially drive up transaction costs. The magnitude of transaction costs are also influenced by factors such as the degree of uncertainty associated with the transaction, the frequency, the strategic significance, and the atmosphere in which it is conducted.

Transaction cost theory assumes that all parties are characterized by opportunistic behavior, bounded rationality, and risk neutrality. These and other key assumptions are illustrated in Fig. 2.13.

Opportunistic parties can be assumed to act solely in accordance with their own interests. This means that if one of the partners has an opportunity to maneuver the other partner to renegotiate terms, he or she will usually do so. Certain information, such as compatibility with other solutions, is likely to be held back. What’s more, opportunistic players do not always fulfill their promises when other parties cannot easily determine compliance (e.g. regarding the quality of a software module). In the software industry in particular, judgments regarding a product’s quality can often only be made after the solution has been deployed—opening the door for opportunism. Bounded rationality in this context means that transaction partners do not have access to all available information (e.g. about a customer’s solvency).

Fig. 2.14 Transaction costs as a function of asset specificity (Williamson 1991, p. 284)



2.4.2 Division of Labor Among Companies: A Transaction Cost Theory Perspective

Probably the most well-known branch of transaction cost theory addresses the question of where an enterprise begins and ends. For simplicity's sake, a clear boundary is drawn between the company (also termed hierarchy in the model) and the market and (as a later addition)—cooperation as a hybrid of market and hierarchy. These three basic types of arrangement are characterized by the following coordination mechanisms: the price for the market, the organization for the company, and a mixture of the two for alliances.

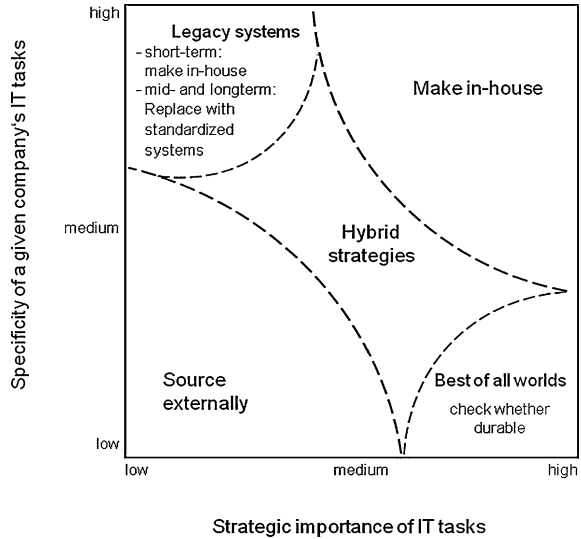
According to the theory, when transactions are not associated with transaction-specific investments (i.e. when they are unspecific), the market is the most efficient coordination mechanism. On the other hand, if the partners become dependent on one another through high transaction-specific investments, they have more incentive to act opportunistically, taking advantage of the other party's dependency. However, this opportunism can be effectively restricted by means of the hierarchy. This means that when specificity is high, hierarchy is the most efficient coordination instrument. Likewise, moderate specificity calls for cooperation as the coordination instrument, as it combines elements of markets and hierarchies.

Transaction cost theory has developed these ideas further, as illustrated in Fig. 2.14. In this representation, when specificity is below s_1 , the market is the most efficient coordination form. When specificity is between s_1 and s_2 , an alliance is the most efficient. And when specificity is greater than s_2 , the company is the coordination instrument of choice.

Figure 2.14 illustrates the transaction costs associated with various organizational forms, at different levels of specificity. However, the exact shape of the curve cannot be determined by analytical or empirical means. This is why it is discussed controversially in the literature.

Initially, transaction cost theory mainly focused on explaining the emergence of enterprises and alliances as alternatives to the market as regulating mechanisms—a macroeconomic question. Today, the theory has become a standard business

Fig. 2.15 Decision matrix for outsourcing IT tasks (Picot and Maier 1992, p. 21)



instrument for determining the boundaries of a given enterprise. It typically addresses make-or-buy decisions at the level of individual tasks, and relationships with the contractor. In the software industry, this topic and its many variations are of great importance—from outsourcing to specialists, to the use of offshoring. We will take a closer look at these questions in Sect. 4.1.

Concerning software enterprises in particular, transaction theory provides valuable insights into the factors of greatest interest to the end customer, especially the business user. In contrast to make-or-buy decisions as mentioned above, this question has long been addressed exhaustively in the literature. For example, simple management instruments were developed that support decision-making concerning outsourcing IT tasks. Figure 2.15 illustrates the tool suggested by Picot and Maier.

In addition to specificity, Picot and Maier also consider the IT tasks' strategic importance—as we already mentioned briefly in the section describing the variables involved in transaction costs. In general, they recommend completely outsourcing only those IT tasks that have a relatively low specificity and a low strategic importance. Network operations fall into this category for many enterprises—which explains why so many companies' foray into outsourcing begins with this task.

2.4.3 Structural Changes to Transaction Costs: The Move to the Middle

We have already shown that transaction cost theory is helpful in the search for the most cost-effective form of organization for a given environment. In the real world, however—particularly in the software industry—the environment is by no means unchanging. New communications technologies and the rise of new nations with impressive IT skills and relatively low labor costs are bringing about radical

change. Against this background, we will sketch the impact of these changes on transaction costs and in consequence, on the selection of the most cost-effective form of organization.

Most market watchers today agree that new communications technologies such as the Internet and IP-based services are leading to a reduction in variable transaction costs and causing the curve in Fig. 2.14 to sink and flatten (Bauer and Stickel 1998). As a result, the point of intersection for the transition to the next form of coordination is moving to the right. This means that moving to cooperative or hierarchical relationships only begins to make sense in economic terms at a higher asset specificity. A pithy term for this trend is “move to the market”. The core proposition is that as a result of new communications technologies, coordination within an enterprise is giving way to cooperative relationships and markets, and that value chains are becoming more fragmented. For the software industry, this would mean that a large number of specialized businesses would arise—a scenario that could be reinforced by the new service-oriented paradigm (see Sect. 4.7).

However, although many companies are now collaborating more intensively than they did in the past, they are often doing so with few partners. This cooperation strategy demands transaction-specific investment, for example for agreeing and implementing special modes of data exchange. In accordance with transaction cost theory, this leads to an increase in asset specificity which would tend to counteract the “move to the market” (Clemons et al. 1993). In summary, both trends taken together mean that cooperation is beneficial, a phenomenon which is referred to as the “move to the middle”.

The new opportunities for outsourcing IT services to low-wage countries, which we discuss in Chap. 4, also play a key role. In terms of classical economic theory, this means that new providers are entering the arena, who will tend to reduce prices on the consumer market or make it attractive to outsource certain tasks that were performed more cheaply in-house until now. At this point, transaction costs must be taken into consideration. A software provider will only outsource services to a software company in India, for example, if the lower development costs resulting from massive wage differentials are not completely offset, and more, by higher transaction costs. Many factors can raise transaction costs, ranging from the greater effort required to develop software specifications and make modifications, to the additional costs associated with supervising the outsourcing provider. Even this brief list of potential issues makes clear that outsourcing software development tasks does not automatically reduce overall costs. Bad decisions can also be made because cost accounting methods typically only show up only a company’s production costs, but not the transaction costs. We will return to this issue in Sect. 4.1 of this book.

2.4.4 Excursion: Intermediaries and Transaction Costs

Transaction cost theory centers around transactions, which we defined earlier as the transfer of usage rights. Taking the transaction as the starting point of analysis, transaction cost theory offers insight into the form of organization that makes most

economic sense in terms of the size of transaction-specific investments. Aside from this, the concept of a transaction is used again and again in discussions of the economic importance and usefulness of intermediaries.

Until now, most intermediaries in the software industry have been marketing agents or distributors sandwiched between the developers and the users of a software product, particularly in business-to-consumer markets. In the near future, the trend toward splitting conventional standard software packages into modules and the emergence of service-based architectures will lead to intermediaries wielding unprecedented influence in other subsectors of the industry, too. Against this background, we will present the basic tenets of intermediation theory below, despite the fact that this is not directly connected to transaction cost theory. We will then return to intermediation theory in the sections on distribution strategy and industrialization.

Intermediaries are companies that do not make products or provide services themselves, but deliver a good or a bundle of goods to those with a demand for it. The most familiar example of intermediaries is retailers who offer consumers a large number of products from various manufacturers at a convenient location for their customers. But intermediaries can also be publishers or television companies that very often do not create content themselves but purchase it from freelance writers or specialized production companies and offer it in the form of a daily newspaper or a TV channel in accordance to customer preferences. We have already given various examples from the software industry. There is a place for intermediaries whenever conducting a transaction through them is more cost-effective than conducting one directly between the supplier and the customer.

Baligh and Richartz's mathematical model makes it possible to assess the effect of an intermediary, at least with regard to the search phase of a transaction (Baligh and Richartz 1967). Let us assume a market with m suppliers and n customers. If a customer wishes to gain an overview of what is on offer, it needs to ask all m suppliers. As this applies for each of the n customers, $m \times n$ contacts will be necessary. However, if an intermediary is involved, each supplier and each customer only needs to register its details with an intermediary, and only $m + n$ contacts are required. Even when n and m are relatively small, this means that employing an intermediary is preferable in terms of the number of contacts required, and this therefore makes good economic sense.

2.5 Software Development as a Principal-Agent Problem

2.5.1 Incentive-Compatible Compensation and Efficient Control

In the development of custom software as well as in implementation projects, maintenance work and the operation of standard software solutions, software companies and customers work closely together. This close collaboration gives

rise to complex dependencies which in economics are known as principal-agent problems. The rest of [Sect. 2.5](#) will present the underlying idea ([2.5.1](#)), show how it can be applied in practice through compensation schemes ([2.5.2](#)) and control systems ([2.5.3](#)).

2.5.2 Principal-Agent Relationships: Definitions and Basic Principles

The principle-agent theory (Spence and Zeckhauser 1971; Ross 1973; Jensen and Meckling 1976) focuses on the division of labor relationship between a principal and an agent. A relationship of this kind exists when, to realize its interests, a principle transfers powers of decision-making and execution to an agent and offers compensation in return.

For our purposes, the relationship between a software company (agent) and a company (principal) that commissions the former is particularly important, where the software company takes responsibility for developing, delivering, implementing or operating software and is paid to do so. A different principal-agent relationship is also of interest to us: if a software company outsources one part of the development process to another company, for example the development of a software module to a business located in a country where labor costs are lower, the software company is then the principal and the outsourcing company in the low-wage country assumes the role of agent (see also [Chap. 4](#)).

The principal-agent theory is based on two core assumptions. First, it is assumed that both the principal and the agent are motivated by utility maximization and systematically use discretionary powers in their own interest (e.g. by over-reporting the programming time taken). Second, it is assumed that the agent has an information advantage with respect to performing the task assigned. The agent generally knows better than the principal, for example, how many person-months a development project will require, and will often not reveal its true interests (e.g. to lock the principal into a long-term relationship). This means that the principal and the agent act rationally in terms of the (incomplete) information available to them, which is termed bounded rationality. [Figure 2.16](#) gives an overview of this basic model.

Opportunistic behavior and bounded rationality in the principal-agent relationship can result in three types of problems which also describe the relationship's structure in temporal terms.

Hidden characteristics arise before conclusion of the contract. Initially, the principal is not fully cognizant of the quality attributes of the agent and the services he or she delivers. This situation involves the risk that the principal could select an unsuitable agent. That is the case when the customer commissions a software company that does not possess the knowledge and skills needed to develop a particular solution.

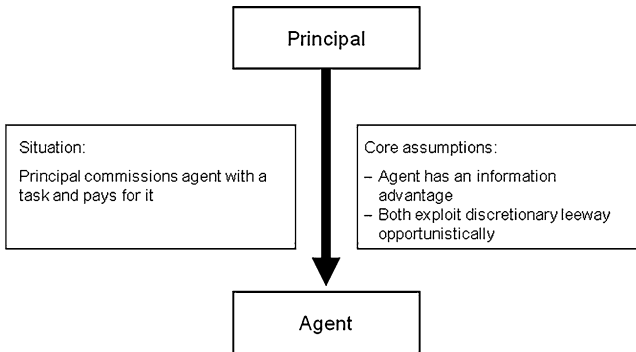


Fig. 2.16 Principal-agent relationship

Hidden action describes the risk that the principal cannot observe or cannot evaluate the agent's actions. In both cases, the principal knows the end-result but has no way of assessing to what extent this is due to the agent's efforts or external factors. As a result, the agent can always exploit his or her freedom to its own ends. For example, a customer will find it very difficult to discover what quality assurance tests have been performed and what the outcome was.

Hidden intention denotes the danger that the principal can become dependent on the agent because it relies on the services the agent performs. Like hidden action, hidden intention only arises after the contract has been signed. The agent can exploit this fact to its own advantage. For example, if a customer concludes an outsourcing agreement, it will automatically be dependent on the outsourcing provider.

Hidden characteristics, hidden action and hidden intention lead to so-called agency costs for the principal. To reduce these costs, the principal can use mechanisms such as compensation plans and control systems. Both these approaches will be described below, and we will provide insight into both quantitative and qualitative research on principal-agent theory.

2.5.3 Incentive-Compatible Compensation Schemes

Performance-based contracts or contracts for services are the basis for compensation of services rendered. The form a compensation scheme takes depends on the type of contract involved. Compensation schemes for both types of contract are described below (Picot and Ertsey 2004).

2.5.3.1 Compensation Based on a Contract with a Defined Result

By signing a performance-based contract, the service provider undertakes to deliver a defined result with clearly specified characteristics in accordance with

agreed terms and conditions. So only the price remains to be set. There are two ways of doing this: The customer can either recompense the provider for the actual costs incurred plus some additional amount (“cost-plus”) or pay a fixed price. With cost-plus contracts, the customer alone bears the project’s cost risk. The provider is in a position to report inflated costs, and even clauses stipulating the disclosure of cost information cannot rule this out completely. A cost-plus contract makes sense when the project specifications are incomplete and the provider is not able to estimate the production costs.

However, if one of these two conditions is not fulfilled, a fixed price agreement is advisable. Under an agreement of this kind, the customer pays a previously agreed fixed price at the end of the project. This means that the provider alone bears the projects’ cost risk. If the costs incurred by the provider are below the agreed price at the end of the project, the provider has realized an (additional) profit. But if the total project costs exceed the agreed price, its profit margin shrinks correspondingly and in the worst case, it might even make a loss. A fixed price certainly gives the provider an incentive to conduct the project efficiently. At the same time, however, there is no motivation to pass on any savings to the customer. We will discuss methods of estimating costs and approaches to pricing custom software projects in [Sect. 3.3.4](#).

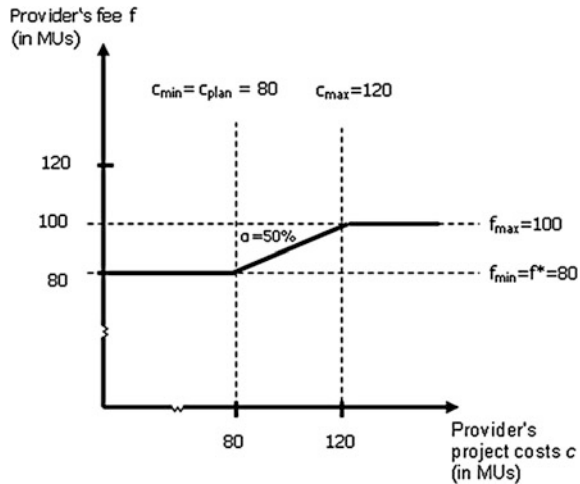
Both types of contract feature a completely one-sided risk situation, with only one of the two parties bearing all of the risk. This gives the other side leeway to act opportunistically. The model presented below describes the distribution of risk for cost-plus contracts. Let us assume that a provider receives compensation e . This comprises a fixed fee f_{\min} and a certain share α of the difference between planned and actual costs $c_a - c_p$, where c_p represents the planned costs and c_a the actual costs. The customer and the provider therefore share the excess costs. If c_a is below the contractually defined minimum c_{\min} , then the provider still receives the minimum amount f_{\min} —thereby realizing additional profit. But if c_a exceeds c_{\max} , the provider still receives the agreed maximum amount f_{\max} —and its profit margin sinks, or it may even make a loss. The compensation scheme, which limits risk for both parties, is as follows:

$$f = \begin{cases} f_{\min} & \text{where } c_a < c_{\min} \\ f_{\min} + \alpha \cdot (c_a - c_p) & \text{where } c_{\min} \leq c_a \leq c_{\max} \\ f_{\max} & \text{where } c_a > c_{\max} \end{cases}$$

Figure 2.17 illustrates this compensation scheme with an example in which c_p and c_{\min} both amount to 80 monetary units (MUs).

Additional development costs up to a maximum of 120 MUs are contributed by both parties equally, i.e., for $c_a = 100$ MUs, the provider receives 80 MUs plus $0.5(100 \text{ MUs} - 80 \text{ MUs}) = 90$ MUs. Project costs in excess of 120 MUs are borne solely by the provider. If the project costs are below 80 MUs, the provider alone profits from the efficiencies achieved.

Fig. 2.17 Example compensation scheme based on a contract with a defined result



2.5.3.2 Compensation Based on a Contract for Services

Under a contract for services, the provider undertakes to perform a particular task such as programming or the provision of a service. If a fixed price is agreed, the provider has an incentive to work cost-effectively and may not focus sufficiently on the quality aspect—after all, it has only agreed to perform a task. This problem can be addressed by means of an incentive-compatible compensation scheme in which the fee is linked to the quality of work performed.

Let us refer to the provider’s performance (e.g. number of lines of code) as p_a . If p_a is below an agreed threshold p_{min} , the provider does not receive any fee—as a deterrent. In the interval $[p_{min}; p_{max}]$, the service provider receives the minimum fee f_{min} and a share of the bonus b which is equal to $f_{max} - f_{min}$. If the performance is greater than p_{max} , the provider only receives the maximum fee f_{max} , so no provider will be motivated to achieve that. If the bonus paid in the interval $[p_{min}; p_{max}]$ is to increase linearly with the performance, the overall fee f is calculated as follows:

$$f = \begin{cases} 0 & \text{where } p_a < p_{min} \\ f_{min} + \frac{p_a - p_{min}}{p_{max} - p_{min}} (f_{max} - f_{min}) & \text{where } p_{min} \leq p_a \leq p_{max} \\ f_{max} & \text{where } p_a > p_{max} \end{cases}$$

In practice, this depends on the ability to measure performance p_a in an objective way. For a software development project, this can be based on lines of code while for network services, service level agreements can be employed.

We will illustrate this scheme by way of a brief example. Let us assume that the performance of a software company is to be measured in lines of code (LoC). Although this indicator is not ideal for various reasons, it can still give a rough idea of the provider’s performance and is used, for example, in models designed to estimate software development effort, such as COCOMO.

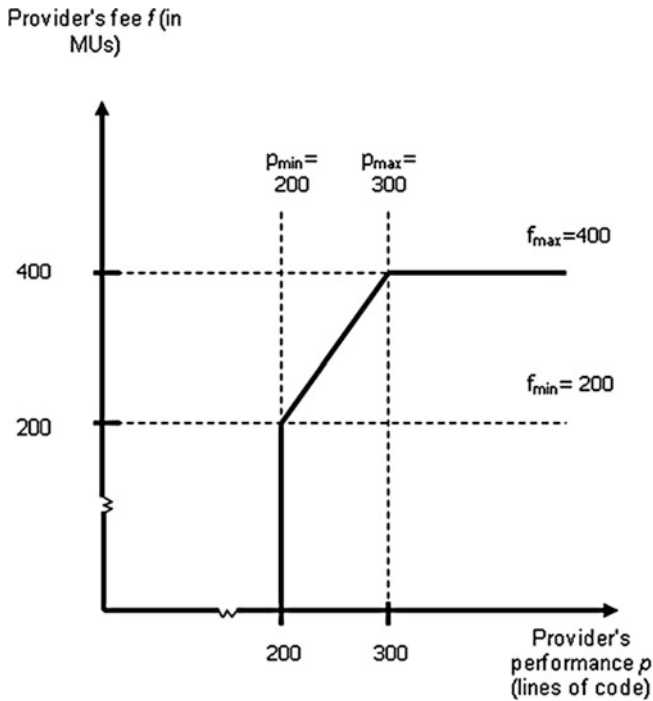


Fig. 2.18 Example compensation scheme on the basis of a contract for services

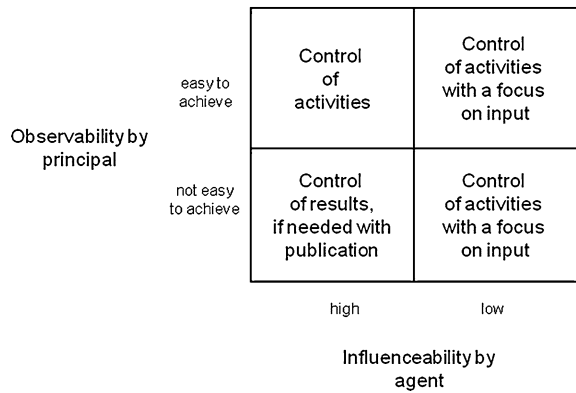
In our example, we assume that the contracted software company is supposed to develop at least 200 LoC and receive a fee of 200 MUs for 200 LoC. Up to 300 LoC, the fee increases linearly up to 400 MUs. Below 200 LoC, the software company receives no fee at all. Above 300 LoC, it always receives 400 MUs. The customer is therefore introducing two separate incentives: at least 200 LoC are required (e.g. to avoid jeopardizing the follow-on project) and up to 300 LoC would be very useful (reflected by the fee for each additional LoC in the interval [200;300] MUs). Figure 2.18 gives an overview of this example.

To conclude, we would like once again to highlight the basic differences between the two compensation schemes presented. Both include the idea of sharing risk within given limits. But the crucial difference is the basis on which the fee is determined: for a contract with a defined outcome, this is the costs incurred by the provider, whereas for a contract for services, it is the provider's performance.

2.5.4 Control Systems

Another way to reduce agency costs besides compensation schemes is to deploy control systems. The design of a control system depends on whether the principal controls only the results produced by the agent or how the agent goes about its work or the input factors such as the development methodology employed.

Fig. 2.19 Various control system designs (based on Hess and Schumann 1999, p. 360)



Research findings on the efficient design of control systems show that the main factors determining the selection of a suitable control system are the agent’s ability to exercise influence and how well the principal can observe what is happening (Ouchi 1977; Picot 1989). Figure 2.19 shows the underlying logic.

Figure 2.19 provides a basis for a general recommendation for software development. Normally, the provider has many ways of controlling the involvement of employees in the project. For example, a custom software provider can choose different project leaders and programmers for a project. It is reasonable to expect that the quality of results will depend on the experience and training of these project members. Given this fact, the agent is able to exert a large influence, so that the left-hand column of Fig. 2.19 applies. However, it is normally difficult for the principal to observe and evaluate the agent’s efforts. Even if the customer understands the software development process per se, perhaps because he himself is a member of an IT department, he seldom has truly reliable information about the concrete status of the project. So in the final analysis, he has no alternative but to limit himself to controlling the results and possibly threatening to acquaint the industry at large with the provider’s failures—this is why reputation is sometimes referred to as the customer’s collateral.

The methods of limiting agency costs discussed above are based on a series of assumptions that simplify the real state of affairs. For example, when formulating the principal-agent problem, whether with the software company as the provider or the customer in the case of sub-contracts, we assumed that the specifications are created once and then used by the provider as a basis for developing the software. A far more realistic assumption is that the customer needs to be involved repeatedly and at different points in the development process. A further potential complication is that the people who conclude the contract and those with the required knowledge and skills may come from different parts of the customer’s organization, typically the IT department and the department that will actually be using the software, which are unlikely to have the same interests. These two influencing factors lead to much more complex principal-agent relationships than those presented here.

Against the background of the economic principles described in the previous chapter, this chapter examines selected strategies for software vendors. The vendor's positioning within the value chain is of critical importance. [Section 3.1](#) initially considers the opportunities and challenges associated with cooperation strategies. In this context, we also look at acquisitions, which play a key role in software markets. In addition, we discuss sales strategies ([Sect. 3.2](#)) and pricing strategies ([Sect. 3.3](#)). We conclude by exploring key management questions concerning the development of software in [Sect. 3.4](#).

3.1 Cooperation and Acquisition Strategies

This section looks at cooperation and acquisition strategies within the software industry. As already described in [Sect. 2.2](#), these strategies are of central importance, in particular against the background of network effects on software markets. Moreover, a single vendor is unlikely to be able to fulfill all customer needs with its own portfolio of products and services. First, we will highlight the general benefits and challenges of cooperation ([Sect. 3.1.1](#)). We will then investigate acquisitions ([Sect. 3.1.2](#)).

3.1.1 Cooperation in the Software Industry

3.1.1.1 Advantages

Cooperation refers here to collaboration between legally independent entities (based on contractual or tacit agreements). We assume that cooperation is for the medium or long term, and requires capital investment on the part of the participants. The key goals are to secure greater efficiency or value added of a kind that would not have been possible without cooperation. Brandenburger and Nalebuff

refer in this context to a “value net” created by this collaboration (Brandenburger and Nalebuff 1996, pp. 16–19). The advantages that can be secured include the following:

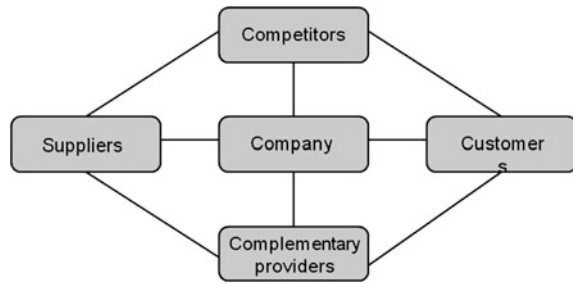
- *Cost savings* can be achieved through economies of scale and economies of scope. Examples include the savings possible by making use of the discounts available when purchasing large volumes or through the shared use of resources, such as warehouses or offices. Moreover, costs can be saved through cooperative planning processes, for example for planning procurement or deliveries (Martín Díaz 2006).
- *Time savings* can be achieved, for example during development work, by combining resources. This can accelerate time-to-market. Development partnerships can also reduce risk. For instance, by sharing the costs of development work, the risks of failure are shared and therefore minimized for the participants.
- Cooperation can also increase the *value of the product or service*. For example, airlines, car rental companies, and hotels form alliances to offer additional services, such as the synchronized provision and return of rental vehicles and the granting of bonus points. Open-source software is also developed through collaborative activities. The more programmers are involved, then the better the quality of the software, at least as a general rule. In his famous article, “The Cathedral and the Bazaar,” Eric S. Raymond states “given enough eyeballs all bugs are shallow” (Raymond 1999).
- Cooperation and acquisitions can grant *access to new markets*. This can involve new geographies, or an extension to the existing product portfolio.

The potential advantages are ultimately reflected either in savings or (directly or indirectly) in an increase in revenues. Dividing up the value added between the partners can be a significant challenge. Many cooperative projects stumble at this particular hurdle—which is only trivial at first glance—even before they really begin. We wish to illustrate this problem using an example of cooperative game theory.

A rich man and a beggar are walking along the road, and both of them discover some money, let us say 100 euros, simultaneously. The challenge for the two of them is to divide up the money in a way that is satisfactory for both. If they manage to find a consensus, each gets to keep his portion of the find. If they fail to agree, both will be left empty-handed.

The problem is easy to understand, but a mutually agreeable solution is difficult to find. Both the beggar and the rich man want to keep as much of the money as possible for themselves. This particular constellation can often be seen in similar form between business partners. The theoretical solution of the problem is complex in mathematical terms, and is based on a variety of utility functions for the two parties (Sieg 2005, p. 181 ff.). On the basis of an assumed linear utility function for the rich man and a logarithmic function for the beggar, it is possible to arrive at a split of approximately 23 euros for the beggar and 77 euros for the rich man. However, this is the theory, and in practice it is very difficult to get the two parties to agree on a split of this kind.

Fig. 3.1 Systematic definition of potential partners



We, therefore, wish to attempt to simplify the matter: we can assume that the allocation of the benefits of cooperation must be Pareto optimal. In other words, the gains must be divided up in a way that ensures that at least one of the partners is better off than before, but none is worse off. If this is not the case, then at least one of the partners will generally not wish to cooperate.

There are pragmatic alternatives, including the following simple approaches (Buxmann et al. 2007):

- The profits of cooperation are divided among n participants, so that each participant receives the n th share of this additional profit.
- The profits of cooperation are divided up in accordance with the pattern of profits prior to the partnership.

In the first instance, the smaller partners gain disproportionately. With the second model, the stronger participants tend to be at an advantage. Even if one or both of these possible models lead to Pareto optimality, this does not mean that the participants would actually come to an agreement. There are a wide variety of possible distribution mechanisms, with an equally large variety of advantages and disadvantages for the individual participants. Whatever happens, each participant will attempt to secure the largest possible slice of the available cake for themselves.

Moreover, any partnership entails a sizeable investment on the part of the participants. This begins with the costs of seeking the right business partner. Then there is the expense, often considerable, of contract negotiations. Negotiations will focus, for example, on the partners' contributions in terms of investments and their share of any profits. Moreover, there will be significant investment in the establishment of a shared infrastructure and in developing skills at the participating organizations (Hirnle and Hess 2006).

In the next section, we wish to concentrate on forms of cooperation within the software industry.

3.1.1.2 Types of Cooperation and Partners Within the Software Industry

First, let us consider who the potential partners for software companies are. In this context, we consider the Brandenburger and Nalebuff model, in Fig. 3.1 (Brandenburger and Nalebuff 1996, p. 17).

In other words, we assume that a company, in the automotive industry, the software industry, or in a different sector, has the four potential partners given above: customers, complementary providers, suppliers, and competitors.

Moreover, partnerships can be better understood by employing the classification created by Ralf Meyer, which differentiates between (Meyer 2008):

- Development partnerships,
- Reseller partnerships,
- Shared revenue partnerships,
- OEM partnerships,
- Referral partnerships, and
- Standardization partnerships.

When defining collaborative relationships, the following questions need to be addressed (Meyer 2008):

- Which partner is to deliver the product to the end-customer (who ships)?
- Which partner determines pricing and discounts (pricing)?
- How is the product to be branded (branding)?
- Which partner has the intellectual property rights to the product (IP)?
- Which partner will post sales revenue for the product (revenue booking)?
- Which partner is responsible for customer contact (customer control)?
- Which partner is responsible for go-to-market activities?
- Does the recipient carry out quality checks (quality assurance)?
- Which partner is responsible for support?

Development Partnerships

A development partnership is a collaborative relationship with the aim of creating new software and service solutions. Software providers can cooperate with any of the partners depicted in Fig. 3.1. The development tasks are shared among the partners, taking on the role of inventors.

A frequent example is the joint development of a product and systems on the part of standard software vendors and their customers. Generally, the aim is to develop a solution that supports industry-specific needs that are not modeled within the standard product. What are the advantages of this type of cooperation? The customer gains access to a software solution tailored to its particular needs. The software vendor gains insight into a particular industry. An example is the development of a solution for scheduling agreements with suppliers, created within the scope of a partnership between SAP and Bosch (Buxmann et al. 2004).

Frequently, as the following example illustrates, joint software development leads to the establishment of a joint venture.

iBS Banking Solution

Within the scope of a dedicated project, CSC Ploenzke created an in-house solution for the mortgages unit of DePfa Bank (since renamed Aareal Bank). The solution adds important components to the standard SAP solution for banks. Its functionality encompasses all lending and deposit-taking, integrated derivatives, and money market and foreign exchange trading. The solution also forms the basis for end-to-end banking management, including risk management, on the basis of the SAP system. Once the project was completed, the participants came to the conclusion that the software solution could be marketed to other companies. The two parties founded a joint venture, with CSC Ploenzke holding a 51 % stake and DePfa the remaining 49 %. There were two reasons for this move. First, there was an issue with the fact that potential customers were DePfa's direct competitors. Second, DePfa did not have sufficient consulting resources to implement the software at other organizations.

Sources www.ibs-banking.com; Sapinfo.net/SAP-BranchenmagazinBanken und Versicherungen, no. 3 March 2001, pp. 20–21.

Suppliers are an additional group of potential partners for joint development activities. In the software industry, these particularly comprise other software companies who can take responsibility for selected development tasks, or freelance contractors who can be incorporated into project teams. Major standard software vendors, for example, frequently work with a large number of software subcontractors. In particular, the trend toward service-oriented architectures (see [Sect. 4.7.2](#)) may facilitate the establishment of further collaborative arrangements at the interface between software vendors and subcontractors. This opens up new opportunities for niche players to offer software as a service (SaaS).

When a standard software provider finds that the costs of developing a particular service exceed the expected value added, the obvious solution is to outsource this development work to a software supplier. This limits the software provider's development risk and the supplier gains an opportunity to integrate its services into the major player's solution. To date, in most actual cases of these types of partnership, the provider and the supplier deal with the customer separately—and send their own invoices. In the future, closer collaboration will be conceivable and worthwhile. The partnership could be set up so that the suppliers will have a share in the revenues. A major challenge here, however, is to find the right formula to divide up the spoils. In addition, both partners must invest in the partnership. Typically, the supplier will have to make the larger investment, especially in the form of training costs. Often, suppliers and development partners must attend—often costly—training sessions organized by the software provider, in order to gain entry to particular partner programs. However, this is an understandable approach from the provider's perspective: For one thing, it ensures that suppliers are familiar with the relevant underlying technologies, and for another, training can generate considerable revenues.

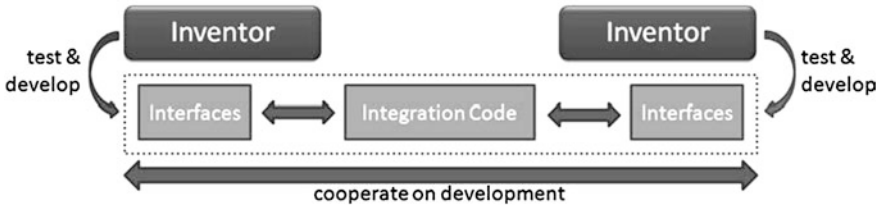


Fig. 3.2 Outline of joint development of integration components (based on Meyer 2008, p. 110)

This type of partnership would also be possible with *competitors*. This relationship is sometimes described as “co-opetition,” because “you have to cooperate and compete at the same time” (Brandenburger and Nalebuff 1996, p. 4). The challenge here is to establish a win–win situation, where both partners profit from the arrangement, although they are and will remain competitors.

The potential advantages of co-opetition are more or less those already outlined in Sect. 3.1.1.1. One obvious motivator of cooperation is the opportunity to share resources. If at least one of the parties has idle development or production resources, while its competitors are operating at close to capacity, it makes sense to share those resources. The best-known examples of this form of cooperation are to be found in the automotive industry. For instance, Daimler and Volkswagen have long collaborated on the development of engines and commercial vehicles, while Porsche and Toyota are working closely together on the development of hybrid drive technology.

DUET is an example of cooperation between two major software providers that are competitors in some areas: Microsoft and SAP. DUET offers users an interface between Microsoft’s Office applications and SAP’s ERP systems. Microsoft and SAP offer competing enterprise software solutions for small and midsize companies, and both hope that the joint solution will deliver a win–win situation.

Payment for and allocation of intellectual property (IP) rights among the parties varies considerably from case to case. Figure 3.2 illustrates a type of development partnership that is often found in the SAP ecosystem. This concerns the joint development of integration components.

Figure 3.3 depicts another type of development partnership, exemplified by the integration of partner products into a SAP solution.

Reseller Partnerships

This model is characterized by a purchaser–provider relationship. In this case, the purchaser incorporates the provider’s solutions in its product portfolio and resells them to its own customers. In other words, the provider produces the software and sells licenses to purchasers, taking on the dual role of inventor and—from the purchaser’s perspective—IP lessor (see Sect. 1.4). The purchaser also occupies the role of IP lessor, as it resells the licenses it has bought. Figure 3.4 provides an outline of a reseller partnership.

Who ships	Pricing	Branding	IP ¹	Revenue Booking	Customer Control	GTM ²	QA ³	Support by
Integr. partner ⁴	Integr. partner	No co-branding	No joint IP	Integr. partner	Integr. partner	Integr. partner	Integr. partner	Integr. partner
SAP	SAP	No co-branding	No joint IP	SAP	SAP	SAP	SAP	SAP

¹IP = Intellectual Property ²GTM = Go-to-market ³QA = Quality Assurance ⁴Integr. partner = Integration partner

Fig. 3.3 Development partnership–parameters (exemplified by an SAP integration project) (based on Meyer 2008, p. 76)

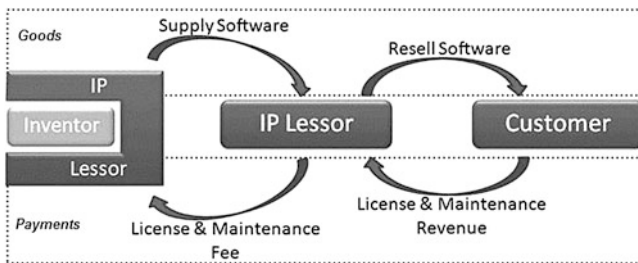


Fig. 3.4 Outline of reseller partnership (based on Meyer 2008, p. 78)

Who ships	Pricing	Branding	IP ¹	Revenue Booking	Customer Control	GTM ²	QA ³	Support by
Resell partner	Resell partner	Both or IP Distr.	No joint IP	Resell partner	Resell partner	Resell partner	Resell partner	Level1,2: Resell partner

¹IP = Intellectual Property ²GTM = Go-to-market ³QA = Quality Assurance

Fig. 3.5 Reselling–parameters (based on Meyer 2008, p. 76)

Reseller partnerships are commonly found in the case of software providers and their sales partners. But they also occur between suppliers and providers.

For suppliers, the potential advantage of this arrangement is that they obtain access to the provider’s customers, without having to establish their own sales organization. While coordinating the partnership involves some additional costs, these are likely to be offset by the increased revenues from the sales channel.

Reseller partnerships are usually organized as shown in Fig. 3.5.

Shared Revenue Partnerships

This type of partnership involves a software provider selling its products through a broker. The latter may also be a platform. Above all, the broker allows the provider to benefit from its strong market position by means of a joint market presence.

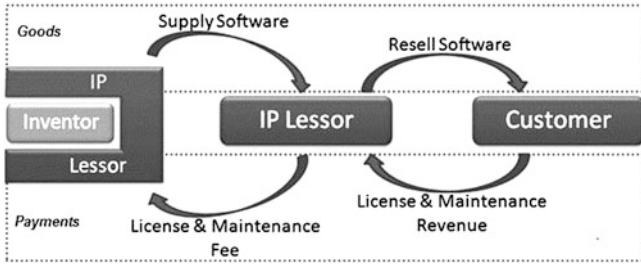


Fig. 3.6 Outline of shared revenue partnership (based on Meyer 2008, p. 86)

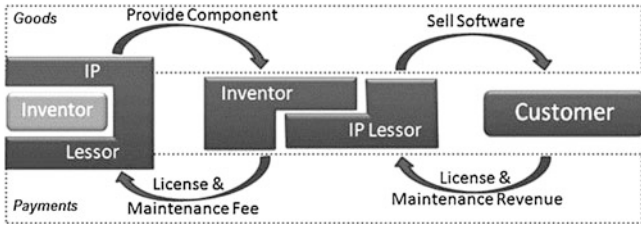


Fig. 3.7 Outline of OEM relationship (based on Meyer 2008, p. 97)

In contrast to resellers, brokers do not assume any proprietary rights to the software products they sell. Instead, they merely recommend the third-party solution to their customers, fulfilling the role of an IP broker. This model is illustrated in Fig. 3.6.

Software distribution via mobile application portals is a current example of this type of cooperation: Software providers—either companies or individual developers—can use these platforms to provide customers with software solutions and/or apps for their mobile devices. The best-known examples are Apple’s App Store and Google’s Android Market.

OEM Relationship

The OEM model is essentially the traditional buyer–supplier relationship, as found in the automotive industry, for example. In other words, a company integrates components or systems developed by suppliers into its own product. In the software industry, OEM relationships involve both suppliers and buyers acting as inventors and IP lessors (Fig. 3.7). In rare cases, buyers can even assemble a solution entirely from supplied components. In this scenario, the buyer’s role as an inventor is limited to the activities required to integrate the components.

This model enables suppliers to benefit from higher sales revenues and greater market penetration. Often, this type of relationship consists of smaller companies providing larger ones with components. But it can also look completely different. For example, SAP frequently acts as an OEM partner by providing its NetWeaver platform to other companies. On this basis, the latter can develop complementary solutions and sell them as a package to their own customers.

Who ships	Pricing	Branding	IP ¹	Revenue Booking	Customer Control	GTM ²	QA ³	Support by
Buyer	Buyer	No co-branding	No joint IP	Buyer	Buyer	Buyer	Supp. ⁴	Buyer

¹IP = Intellectual Property ²GTM = Go-to-market ³QA = Quality Assurance ⁴Supp. = Supplier

Fig. 3.8 OEM relationship–parameters (based on Meyer 2008, p. 76)

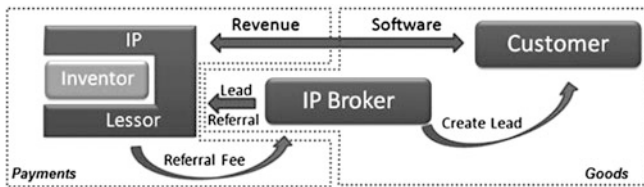


Fig. 3.9 Outline of referral partnership (based on Meyer 2008, p. 93)

Who ships	Pricing	Branding	IP ¹	Revenue Booking	Customer Control	GTM ²	QA ³	Support by
Ref. recipient ⁴	Ref. recipient	Ref. recipient	Ref. recipient	Ref. recipient	Ref. recipient	Ref. recipient	Ref. recipient	Ref. recipient

¹IP = Intellectual Property ²GTM = Go-to-market ³QA = Quality Assurance ⁴Ref. recipient = Inventor resp. IP Lessor

Fig. 3.10 Outline of referral partnership (based on Meyer 2008, p. 76)

From the buyer’s perspective, the main advantage of the OEM model is that purchasing components and solutions cuts development time, costs, and risks. Suppliers can deliver cost savings because they generally sell their products to multiple buyers, enabling them to distribute development costs across several parties.

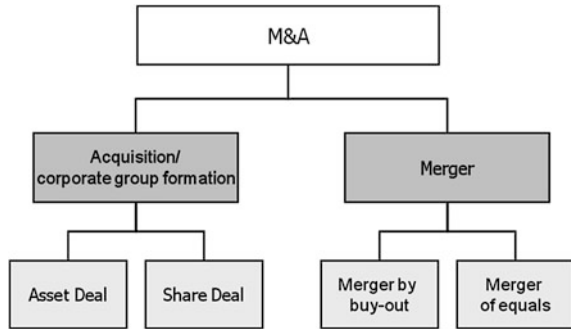
A typical OEM relationship is organized as shown in Fig. 3.8.

Referral Partnerships

Like shared revenue models, referral partnerships involve an IP broker (Fig. 3.9). In the following, we describe the latter as the “referral provider.” This partner sells information about potential customers (leads) to the referral recipient (IP lessor). In return for this information, the IP broker receives a combination of a fixed fee and a share of the resulting revenues. As the referral provider mostly has no direct influence over these customers, but merely passes on their contact details, the share is generally a lot smaller than in the shared revenue model.

Figure 3.10 shows how this partnership is organized.

Fig. 3.11 Types of mergers and acquisitions (Wirtz 2003, p. 13)



Standardization Partnerships

Standardization partnerships are somewhat different: As a general rule, no money changes hands between the parties. However, they are an important element in many software and IT enterprises' strategies. These companies frequently form strategic alliances. Mostly, their aim is to establish standards—or to prevent the widespread adoption of standards that would benefit potential competitors.

The working groups of standardization organizations, such as the World Wide Web Consortium, frequently facilitate this type of cooperation. Participating companies aim to give input into the technical specifications of the new standard. But they also know that they need to be inside the tent to ensure that their own or preferred standard is chosen over others, possibly favored by competitors. We discussed the importance of setting standards in some detail in [Sect. 2.2](#).

3.1.2 Mergers and Acquisitions in the Software Industry

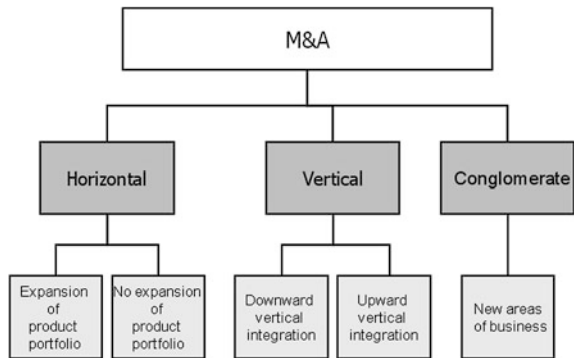
In this section, we will focus on mergers and acquisitions (M&A) in the software industry. These play a particular role in the software industry, because network effects make the size of a provider and its network a crucial competitive advantage. First, we will consider the various forms of M&A ([Sect. 3.1.2.1](#)). This will feed into an investigation of the various motives for acquisitions ([Sect. 3.1.2.2](#)), before we turn to the consolidation trend in the software industry ([Sect. 3.1.2.3](#)). Finally, we will analyze the success of M&A in the software industry ([Sect. 3.1.2.4](#)).

3.1.2.1 Forms of M&A

There is a multitude of definitions surrounding M&A. We do not intend to go into them here. However, a common definition is that acquisitions involve at least one company relinquishing its financial—and possibly also legal—independence (Wirtz 2003, p. 15).

A distinction can be made between M&As, as shown in [Fig. 3.11](#). Mergers entail the fusion of two independent companies into one legal entity. In other

Fig. 3.12 Forms of M&A
(Wirtz 2003, p. 19)



words, both parties surrender their legal independence, although they can opt to form a new legal entity or be incorporated into an existing one. In acquisitions, by contrast, one company is integrated into a corporate group—this does not necessarily involve a legal fusion.

Another distinction can be drawn between horizontal, vertical, and conglomerate (also known as diagonal) M&A (Wirtz 2003, p. 18 ff.).

Horizontal M&A involve companies in the same industry at the same stage of the value chain. They are generally intended to increase competitiveness and realize synergies in the form of economies of scale and/or economies of scope. Examples of horizontal mergers in the software industry include the acquisition of PeopleSoft by Oracle (see Sect. 3.1.2.3) or Symantec’s purchase of storage and security solution provider Veritas, for some \$13.5 billion (Parbel 2005).

Vertical M&A combine companies on different levels of the value chain. In other words, an enterprise joins forces with another that is directly upstream or downstream. This is also known as upward or downward vertical integration. The aims behind vertical M&A are cutting transaction costs, improving planning across the value chain, and better access to procurement (in the case of downward vertical integration) or purchasers (in the case of upward vertical integration). The expansion of Software AG in Latin America is an example of this type of deal. In 2005, the German provider acquired APS Venezuela and five affiliated companies in Panama, Costa Rica, and Puerto Rico. Previous to this, APS Venezuela had been Software AG’s sales partner and an established distributor of transaction systems for major accounts in the financial, manufacturing, oil and mining industries, and the public sector. The move was intended to strengthen Software AG’s presence on the Latin American market.

Diagonal or conglomerate M&A bring together companies from different industries or segments, enabling the penetration of new markets. These generally result from a diversification or expansion strategy. For example, this is the strategy followed by Infor. Its sales revenue of approximately \$2.1 billion and headcount of more than 8,000 make it one of the world’s largest software companies. Infor’s strategy aims for rapid growth by buying up a number of smaller software providers from a variety of fields. In contrast to many other acquisitions in the

industry, those made by Infor are not primarily driven by migration strategy: the company does not intend to integrate the acquired applications into a single end-to-end solution. In addition to licensing, Infor quickly generates income from the target companies' on-going service contracts. Moreover, it can take advantage of cost-cutting opportunities such as tighter cost management or by consolidating corporate or administrative functions.

Figure 3.12 illustrates the various forms of M&A.

In the following section, we will look at the motivations that prompt management to acquire another company or sell their own.

3.1.2.2 Motivations for M&A

As M&A can affect every aspect of the companies involved, any examination should theoretically address the viewpoints of all relevant stakeholders. This would include shareholders, management, employees, suppliers, customers, competitors, and even society itself, as M&As frequently cause significant changes in the labor market.

In the following section, however, we will concentrate on a brief outline of the reasons from management's perspective. These can be grouped into strategic, financial, and personal motivations (Wirtz 2003, pp. 57–76).

Strategic motivations are generally about realizing synergies and can be divided into

- Market motivations,
- Performance motivations, and
- Risk motivations.

Market motivations relate to both the procurement and sales sides. For example, the deal may increase negotiating power over mutual suppliers, as larger quantities are at stake. In addition to procurement, boosting sales is another key motivation for M&A. By combining their sales activities, the companies involved can realize synergies and gain competitive edge. A stronger position on the sales market can bestow greater influence over prices, and can even help to squeeze competitors out. Finally, an acquisition can also offer growth potential, for example in the form of access to new regional markets.

Synergies may also be created by pooling resources and skills. *Performance motivations* are the expectation of synergies in corporate activities such as research and development, procurement, production, marketing. For one thing, the companies' technologies and expertise can be combined to produce new or better quality products and services. In the context of software, the development of integrated systems is conceivable. In addition, the deal could lead to a better utilization of existing development resources.

Risk motivations are most commonly found in relation to M&A activities driven by a diversification strategy. For example, a company may see risks in its dependence on a particular product or the development of a particular industry.

Table 3.1 Comparison of M&A activities by industry, 2009 (USA) (Mergerstat Free Reports 2009)

Rank	Industry	Deals	Value in millions of \$
1	Drugs, medical supplies, and equipment	319	219,089.7
2	Computer software, supplies, and services	1338	50,911.6
3	Brokerage, investment and management, and consulting	521	47,170.5
4	Energy services	83	42,586.3
5	Transportation	81	29,626.0
6	Banking and finance	272	28,632.9
7	Food processing	83	22,665.7
8	Miscellaneous services	816	19,636.7
9	Broadcasting	73	19,069.0
10	Chemicals, paints, and coatings	106	14,060.0
Total		3692	493,448.4
Top 10			

Expanding the product portfolio or tapping into new industries by means of an acquisition is seen as a way of reducing these risks.

It should be noted that the strategic motivations listed above also apply to cooperation in general, and should be regarded as supplementing the list of advantages of cooperation in [Sect. 3.1.1.1](#).

In addition to the above-mentioned market, performance and risk motivations, there may also be *financial motives* for a merger or acquisition. The most powerful motivation is generally raising profitability by generating profits or by taking advantage of tax losses carried forward. These options will be based on considerations relating to developments on capital markets, balance-sheet optimization, and taxation.

Moreover, management may also have *personal motivations*. A number of explanations have been put forward, including managers overestimating their own abilities and empire-building, as the real reasons behind some acquisitions (Wirtz 2003, pp. 57–76).

3.1.2.3 Consolidation Tendencies in the Software Industry

We have already noted a number of times that, due to network effects, software markets are governed by the winner-takes-all principle. M&A help fuel this tendency toward the establishment of monopolies. Their significance in the software industry is highlighted in the following table, which compares M&A activities across industries in 2009. Out of 49 industries studied, the tables show the top 10 on the US (Table 3.1) and European markets (Table 3.2). In the USA, the software industry occupies second place, in terms of transaction volume. By way of comparison: in 2006, the software industry was ranked sixth (Buxmann et al.

Table 3.2 Comparison of M&A activities by industry, 2009 (Europe) (Mergerstat Free Reports 2009)

Rank	Industry	Deals	Value in millions of \$
1	Drugs, medical supplies and equipment	71	25,690.5
2	Brokerage, investment and mgmt., and consulting	86	22,535.7
3	Food processing	20	19,939.3
4	Insurance	35	12,560.7
5	Broadcasting	18	8,955.2
6	Computer software, supplies and services	204	8,755.1
7	Beverages	22	7,059.6
8	Mining and minerals	16	6,044.5
9	Electrical equipment	19	6,040.5
10	Miscellaneous services	144	4,587.6
Total		635	122,168.7
Top 10			

2008a). If we look at the number of deals, though, the software industry topped the rankings in both 2006 and 2009. In Europe, the software industry is sixth in terms of transaction volumes. But with respect to the number of deals, the industry comes out on top here as well.

A well-known example of the consolidation trend is the ERP software segment, most notably the takeover strategies pursued by Oracle. In recent years, Oracle has attracted attention for its many M&A activities.

On June 6, 2003, just 4 days after PeopleSoft disclosed its intention to acquire competitor J. D. Edwards, Oracle announced its own plan to buy up PeopleSoft for \$5.1 billion. The acquisition of J. D. Edwards would have made PeopleSoft the second largest provider of enterprise software after SAP. Oracle's announcement triggered a 19-month long takeover battle. Finally, Oracle emerged victorious, securing its rival for some \$10.3 billion. This made it the industry's largest merger to date. Oracle was restored to its runner-up position on the enterprise software market, and was able to edge closer to market leader SAP.

Oracle promised not only to continue supporting PeopleSoft and J.D. Edwards' software until 2013, but also to develop it further. At the same time, it released a new, integrated software solution called "Fusion," combining all product lines. This initiative must have required a significant effort on Oracle's part; Fusion's R&D team is already one of the world's largest.

Oracle then added one of the leading providers of CRM software to its holdings, acquiring Siebel for around \$5.85 billion. This intensified the wave of consolidations on the ERP software market.

Recently, Oracle purchased Sun Microsystems, giving it a foothold on the hardware market.

A hotly debated theory posits that M&A activity upsets the market's equilibrium: it essentially forces other organizations to make their own acquisitions, in order to preserve their long-term independence on the market and avoid becoming a takeover target themselves (Hutzschenreuter and Stratigakis 2003).

In the software industry in particular, network effects mean that size is in itself an advantage for providers, giving them an edge over the competition. This was the reason given by Oracle CEO Lawrence Ellison, who stated that the acquisition of Siebel would “strengthen our number one position in applications in North America and move us closer to the number one position in applications globally.” Moreover, in this case, Oracle also had the opportunity to offer its customers an end-to-end integrated solution.

A similar consolidation trend can be seen on the market for office software. A few years ago, there were a number of viable options, such as Lotus 1-2-3 for spreadsheets or Word Perfect for word processing. Now, Microsoft has a virtual monopoly over this market. The open source community is the only remaining serious competitor (see [Chap. 7](#)).

Even though monopolistic structures are often regarded as disadvantageous for the buyers, their impact on network markets is a matter of debate. On the one hand, of course, customers are far more dependent on the provider. Theoretically, this would enable the provider to raise prices, which—as shown above—has not happened in the case of Microsoft. An additional drawback for users is that monopolies tend to produce fewer innovations. On the other hand, monopolistic structures have the advantage of avoiding incompatibility issues. This is the line taken by Stanley Liebowitz and Stephen Margolis in various articles (e.g., Liebowitz and Margolis 1994, 2001).

Empirical studies show that the wave of consolidations in the software industry is primarily being driven by leading players (Friedewald et al. 2001). One response open to smaller software providers is to cooperate with other software companies; another is to concentrate on specialist or niche solutions.

3.1.2.4 Factors Determining the Success of M&A Activities in the Software Industry

On software markets, M&As give companies the opportunity to gain strategic competitive advantage. As previously explained at some length, this is reinforced by the fact that software providers operate on network effect markets. But do acquisitions really increase the value of the purchaser's company? This question was addressed by Izcı and Schiereck in an empirical study (Izcı and Schiereck 2010).

The dataset comprised 81 international acquisitions in the enterprise software industry between 2000 and 2007. All of these transactions were worth at least \$50 million and the purchaser acquired at least 20 % of the target company. Both the purchaser and the target were listed companies, whose stock prices could be tracked between 230 days before and 30 days after the announcement of the deal.

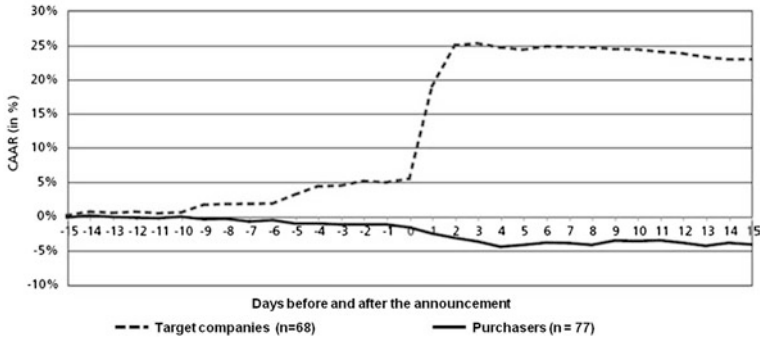


Fig. 3.13 Cumulative average abnormal return (CAAR) for purchaser and target companies (Izci and Schiereck 2010)

The investigation was based on the analysis of abnormal returns. These are the difference between actual return and the “normal” return that would have been expected had there been no M&A announcement. The abnormal returns for the purchaser and target were calculated and cumulated over a period comprising up to 15 days before and after the M&A announcement.

The results showed that target companies obtain a positive cumulative abnormal return (CAR), while the purchasers’ value tends to dip slightly. The CAR experienced a statistically significant rise or fall, particularly when the announcement was made and a few days following it, and the trend continued throughout the observation period thereafter (see Fig. 3.13).

In a second step, the researchers looked at the factors affecting the development of the stock price, and therefore the success of the purchaser. On the basis of a multivariate regression, they found no evidence that acquiring a company from the same industry or segment or in the international environment had the expected positive effect (Izci and Schiereck 2010).

While the analysis did show a small positive correlation, it was not statistically significant. Similarly, the percentage of stock acquired and transaction volume had no meaningful effect on the purchaser’s value. The only factor that showed a slightly positive impact was cash rather than stock purchase: Buying with cash had a positive effect on the stock market valuation.

By contrast, there were significant negative impacts in relation to the relative size and stock market performance of the target company. The stock market reacts unfavorably to major acquisitions, speculating that the integration costs incurred will outweigh the potential synergies (Loefert 2007, p. 163; Izci and Schiereck 2010).

The results of the stock market study show that M&A transactions in the enterprise software industry add value (at least initially) for target companies only, while purchaser companies are at a disadvantage, in terms of shareholder value.

What is surprising is that overall high risk, in what is an innovative and dynamic segment, does not appear to be responsible for the negative valuation of the purchaser. Cross-border or diversifying transactions are not regarded as

particularly negative. Instead, it is cultural challenges that are likely to be punished by the stock market. To earn a positive reaction from the markets, enterprise software providers will have to convincingly demonstrate that the integration of their major competitors will result in strong growth without risking synergies. The alternative—foregoing acquisitions altogether—is not a realistic option in a rapidly consolidating network market like the software industry.

3.2 Sales Strategies

Sales strategies involve decision-making around the provision of goods and/or services to companies in the downstream value chain. This comprises activities that directly address prospects, and sales logistics activities (Homburg and Krohmer 2006, pp. 864–866).

This section discusses channel management activities; sales logistics is about ensuring that physical products reach the end-customer, and is therefore not relevant to software products.

In channel management, the following issues are of central importance:

- How the sales system is structured?
- How relationships with sales partners and key accounts are managed?
- The use of Key Performance Indicators (KPIs) systems in sales management and
- How sales activities are organized?

3.2.1 Structuring of Sales Systems: Organization and Sales Channels in the Software Industry

Structuring a sales system involves taking decisions about the sales organization and sales channels.

Let us begin by looking at the sales organization. This can generally be structured according to:

- Region,
- Industry,
- Product as well as
- New and existing customers.

Structuring sales by region generally means by continent, country or federal state. The advantage of this approach is that it ensures a measure of geographical proximity to the customer.

But sales can also be organized by industry. The advantage here is that the sales professionals have the relevant industry-specific skills and ‘speak the customer’s language.’ One drawback is that they typically spend more time traveling.

Structuring by product is the prime strategy of software companies that have a broad range of products and whose sales activities rely on product-specific skills.

The sale of complex SCM or CRM systems is an example of this. Against this background, many software companies structure their sales and consulting business not only by region, but also by product. In the SCM and CRM field, for example, many software companies have experts who address highly specialized issues in worldwide projects, such as optimization algorithms for the implementation and use of SCM solutions.

It makes sense to structure the sales organization in terms of new and existing customers because these two target groups require different types of sales employees, who can be labeled ‘hunters’ and ‘farmers.’ As the name suggests, the hunter type is best suited to winning new customers and selling them a new product or solution. The farmer type, on the other hand, feels more comfortable working with customers with whom he has built up a relationship over a long period of time. This type should, therefore, be deployed in sales to existing customers.

The above criteria can be combined, of course. It is common practice, for instance, to create roles defined by product and region. This leads to one department being responsible for specific continents and products. The advantage is that the unique characteristics of specific regions and products are fully taken into account, but the disadvantage is that it is not always clearly defined who is responsible for what.

In the following, we will look at how to structure *sales channels*. The most basic decision to be made is whether to choose direct or indirect sales. Indirect sales are when sales activities are performed by third parties, such as VARs and system integrators. With direct sales, these tasks are carried out in-house (Homburg and Krohmer 2006, pp. 873–877).

When choosing between direct and indirect sales, businesses should take into account both efficiency and effectiveness considerations. In terms of efficiency, they should evaluate the two options in light of the associated transaction costs (see Sect. 2.4), as using sales partners can lead to savings in this area. Those savings are comparable with a trade margin in license sales. Effectiveness considerations can relate to customer service quality, for example regarding geographical proximity or specialization, or customer allegiance. For instance, VARs and systems integrators might concentrate on specific industries and develop deep skills. In addition, businesses in fast-growing sections of the software industry tend to work with sales partners, because they could not achieve rapid growth without them.

We will now look at some factors that can influence the advantages of direct versus indirect sales. There is an assumption in the marketing literature that a high specificity and product complexity would favor direct sales (Homburg and Krohmer 2006, p. 874).

But does this necessarily apply to the software industry? With regard to specificity, the assertion is correct. A provider of custom software, i.e., specific

solutions, is not likely to sell those solutions via indirect channels. So, indirect sales are only an option for standard software providers. However, these products are often extremely complex. As we saw in [Chap. 1](#), this applies especially to ERP systems, considering the huge amount of customization work required during implementation projects. Nevertheless, standard software providers frequently opt for indirect sales. But this route calls for sales partners with a correspondingly high level of (often industry-specific) expertise. So, as far as the software industry is concerned, high specificity signals a tendency to use direct sales. A further advantage of direct sales, of course, is that it enables companies to build customer allegiance and intimacy.

The choice of direct versus indirect sales will also depend on how many potential customers there are. The advantages of indirect sales tend to increase with the number of customers. Again, this can be explained in terms of the transaction costs. As we saw in [Sect. 2.4.4](#) in the discussion of intermediaries, savings potential rises with the number of market participants.

Now, one might argue that the availability of the Internet and the characteristics of software as a good would favor direct sales. Why would the provider not simply sell the software direct to customers over the Web? This is no problem when the product is relatively easy to understand, such as antivirus software. But it is quite a different thing when it comes to implementing a complex solution, one with comprehensive supply chain management functionality, for example. In this case, the customer is certain to require a good deal of advice and support.

We underlined the international nature of the software industry at the beginning of the book. This affects the structure of sales in various ways. A study by Lünendonk showed that companies generally conduct indirect sales in other countries via a subsidiary. The second most common option is the use of collaboration partners, such as IT consulting companies, VARs and system integrators (Lünendonk 2007, p. 57).

However, as a rule, providers do not have to opt for exclusively direct or only indirect sales. They can often combine the two. SAP's sales strategy is a case in point. SAP divides up the market and as a result, differentiates its sales strategy in terms of

- Global Enterprises with at least 2,500 employees,
- Local Enterprises with 1,000–2,499 employees,
- Medium Enterprises with 100–999 employees, and
- Small Enterprises with 1–99 employees.

In a presentation to investors in January 2007, SAP offered the following market breakdown:

- 20,000 companies in the Global Enterprise segment,
- 1.3 million companies in the Local and Medium Enterprise segment,
- 55.4 million companies in the Small Enterprise segment.

Table 3.3 Availability of IT skills in midsize companies

Resources	Lower mid-market	Upper mid-market
IT department	None	In-house
ERP software skills	None	Yes
IT budget	None	Yes
Software decision makers	Senior management	IT department
Process complexity	Low	High
Number of users	Small	Large

SAP sells direct to global enterprises, i.e., it does not involve any partners in the sales process. The installed base in this segment ensures repeat sales and a steady stream of license income through upgrades, new releases, and maintenance.

As SAP already has a very large market share among the top 500 firms, its future growth will depend mainly on establishing a strong position among small and midsize enterprises. In line with this objective, the company launched the SaaS solution, SAP Business ByDesign. A major challenge is that the market segments targeted differ markedly in terms of structure. Moreover, the various industries represented by midsize companies make quite different demands on business software. As a result, SAP's Sales and Consulting units responsible for the lower mid-market are locally organized, whereas those for the upper mid-market are active throughout the whole country or even globally. These midsize enterprises can be very different with respect to the IT skills at their disposal, as the parameters in Table 3.3 describe.

SAP aims to acquire new customers in these segments by means of preconfigured industry-specific solutions with attractive entry-level pricing. In collaboration with its partners, the software giant is utilizing a new sales and implementation strategy: the try-run-adopt model, which allows companies to try out the software free of charge. In addition, the solutions are available as a SaaS offering (see Chap. 6). The first SAP partners began to market these industry solutions as packages in late 2006. With SAP's support, they offer customers a bundle comprising licenses, maintenance services, implementation, and the day-to-day management of SAP systems at a monthly rate that is currently well below € 200 per user. TV commercials have been used to promote this model.

When determining the shape of their sales system, providers must also decide on

- The length of the distribution channel,
- The width of the distribution channel, and
- The width of the sales system (Homburg and Krohmer 2006, pp. 877–884).

The length of the distribution channel indicates how many sales partners there are between the provider and the customer. In the software industry, multitier

distribution channels make little sense. The advantages to be gained from them relate mainly to logistics, particularly warehouse storage, which are of little relevance to the software industry.

The width of the distribution channel expresses the number of sales partners a provider works with. In principle, it makes more sense to work with a small group of partners when the products sold are complex and of high value. As mentioned earlier, this is why software companies provide training to assure the quality of their sales partners—although another motivation is to generate sales in this sector themselves.

The width of the distribution system describes whether a product is distributed over just one channel or several. A single-channel system is one in which the product can only reach the customer by one channel. With a multi-channel system, the provider employs several channels to the customer. The music industry is one example of this approach: albums are sold via conventional brick-and-mortar retailers, online stores, or digitally via distributors such as Apple iTunes or Musicload.

Key goals of a multichannel system are broad market coverage and the ability to reach different customer segments. A core challenge is to create channels that complement and do not cannibalize each other. For instance, a publisher could offer subscribers of its daily newspaper various online functions that complement the print edition, such as search functionality or multimedia content like documentary films on featured topics. In the software industry, relatively self-explanatory products are sold both on the Internet and through intermediaries. However, cannibalization is less of a problem than in the music industry or in publishing, as software providers generally have no preference as to whether their sales are generated by direct or indirect channels.

When designing distribution systems, businesses need to clearly define the target groups and the tasks of the various distribution channels.

3.2.2 Organization of Relationships with Sales Partners and Key Accounts

We will now turn our attention to a software provider's relationship with its sales partners and key accounts. The term 'key account' denotes customers—usually enterprises—who are particularly important to the provider and who are therefore offered special deals or services.

Let us begin by looking at the provider-sales partner relationship. As in a supply chain, this relationship is based on the purchase by sales partners of the provider's goods and services—in our case, software licenses—in order to sell them on to their customers. The following section illustrates SAP's sales partnerships.

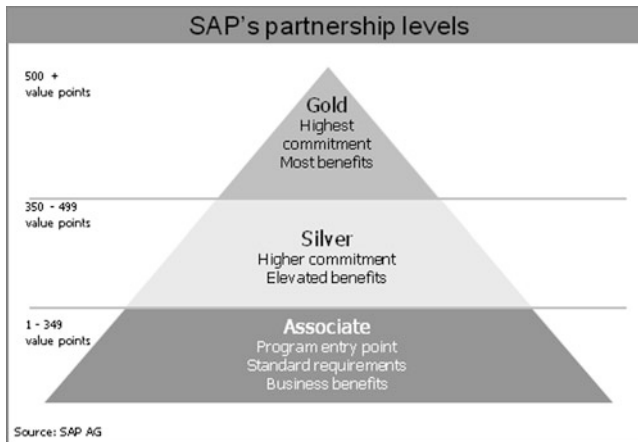


Fig. 3.14 SAP's partnership levels

Sales partnerships in the SAP space

Among SAP's partners, the mid-market is fiercely contested. To provide transparent information about the quality of its partners, SAP introduced a new method at the 2005 SAPHIRE conference, whereby partners are granted a particular status (associate, silver or gold) via a scoring system known as Value Points (see Fig. 3.14). This system evaluates both the partner's skills and its portfolio. While sales success is the main criterion, customer satisfaction, training activities, the development of industry-specific solutions and add-ons also earn companies bonus points. According to the Value Points system, the score required for a particular status must be achieved over the past four quarters. Gold status is worth attaining for several reasons: partners with this status enjoy the biggest discounts on SAP licenses, which means they can achieve the highest margin when they resell them. An immaterial benefit is that the partner has close ties with SAP. Ultimately, the partner status reflects the development of the business relationship between SAP and the partner. A high status is bestowed in recognition of the partner's investment in the skills and resources needed to develop and implement SAP solutions.

SAP also provides partners with an extensive e-learning portfolio, and offers them sales and product training, which again earn them value points. Other concrete support includes quarterly assessments and reviews, strategy workshops, and joint marketing activities.

Key account management is crucial for software providers in the narrow and in the wider sense. That is why it is not unusual for executive board members or senior managers to be involved in this task. Managers often draft internal key account development plans that set out what revenues the company wishes to generate with which solutions. The discounts granted to key accounts, e.g., on software licenses, are normally considerably greater than regular discounts.

A primary goal of key account managers—regardless of the person’s level in the hierarchy—is to be integrated in the customer’s strategic planning. This can include jointly planning updates to a new release or the implementation of new, innovative technologies. Coordination can also be worthwhile on an operational level, e.g., regarding the customer’s spending plan. To help key account managers with their tasks, some CRM systems enable them to create customer maps that include information about which customer employees are well-disposed to the provider and which are not. This information can be particularly vital when problems need to be communicated.

Organizing regular events for key accounts is another good way of improving communications and fostering customer allegiance. These can include invitations to sport and/or cultural events.

Before we examine the organization of sales processes, we would like to start by offering some thoughts on the management of sales activities. Performance measurement systems can be an important tool for this purpose.

3.2.3 Key Performance Indicators as a Sales Performance Management Tool in the Software Industry

3.2.3.1 Sales Performance Management

Sales performance management is the targeted management and coordination of a company’s sales activities. We believe that performance management systems must do more than simply monitor performance by comparing target to actual figures. In a broad field of the literature, the main task of a sales performance management system is seen as the coordination of the management system. A coordination task always involves producing and utilizing decision-relevant information for the purpose of managing a business or part of a business—in our case, sales activities. A wide range of analysis tools is available to help supply pertinent information (Fig. 3.15).

In this section, we will restrict ourselves to showing how performance measurement systems are used, because this allows us to identify some special features of sales activities in the software industry. The use of other tools, by contrast, does not differ significantly between industries.

Analysis tool \ Use	ABC analysis	Portfolio analysis	Cost-performance analysis	Investment analysis	KPIs/ performance measurement systems
Provision of information	X	X	X	X	X
Planning	X	X	X	X	X
Monitoring			X		X

Fig. 3.15 Selected analysis tools for sales performance management (Homburg and Krohmer 2006, p. 1214)

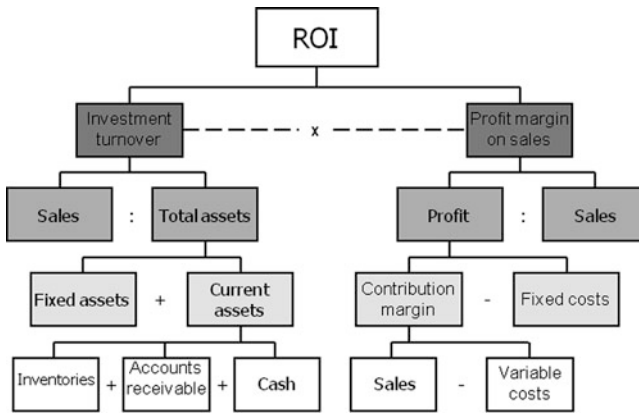


Fig. 3.16 The DuPont ROI model (Küpper 2008, p. 369)

3.2.3.2 Use of Performance Measurement Systems in the Software Industry

Generally speaking, the purpose of KPIs is to evaluate something in quantitative terms. In the business world, performance indicators are used to provide management with decision-relevant information, usually for planning and monitoring. For example, KPIs can be used to specify targets for a given period, and at the end of that time to determine to what extent they have been met. If the actual figures differ from the targets, corrective action must be taken.

A variety of performance measurement systems has been developed, and they include many indicators that are related to each other. A classic example is the DuPont Return on Investment model shown in Fig. 3.16.

Modern management systems, such as the Balanced Scorecard, are also based on performance indicators as a tool for managing performance.

While the performance measurement systems discussed here are largely application independent, Homburg and Krohner developed a classification of KPIs

	Effectiveness	Efficiency
	Category I	Category II
Potential-related KPIs	For example <ul style="list-style-type: none"> • Customer satisfaction • Brand image • Provider's price image • Awareness of offering • On-time delivery 	For example <ul style="list-style-type: none"> • Number of contacts gained / promotion costs • Customer satisfaction with sales support / sales support costs • Customer satisfaction with service level / sales logistics costs
	Category III	Category IV
KPIs for market success	For example <ul style="list-style-type: none"> • Number of customer inquiries • Total number of customers • Number of new customers • Number of lost customers • Number of customers won back • Market share of a product • Price level achieved on the market • Customer allegiance 	For example <ul style="list-style-type: none"> • Number of customer inquiries per order • Number of customer visits per order • Number of quotations per order (hit rate) • Number of successful new product introductions (success and flop rate) • Number of new customers gained / costs of activities for direct communication
	Category V	Category VI
Economic KPIs	For example <ul style="list-style-type: none"> • Revenue • Revenue related to product or product group • Revenue related to customer or customer group • Revenue due to special offer promotions • Revenue due to direct communication activities 	For example <ul style="list-style-type: none"> • Profit • Profit margin • Customer profitability • Revenue due to discounts/costs in the form of lost revenues • Revenue due to participation in trade shows/cost of participation in trade shows

Fig. 3.17 KPIs for performance management in marketing and sales (Homburg and Krohmer 2006, p. 1234)

for sales and marketing. They differentiate between potential-related, market-related, and economic KPIs on the one hand, and effectiveness and efficiency-related KPIs on the other (see Fig. 3.17).

Although the above KPIs relate to marketing and sales, they are industry-independent, i.e., they are as relevant to the software industry as to any other. The

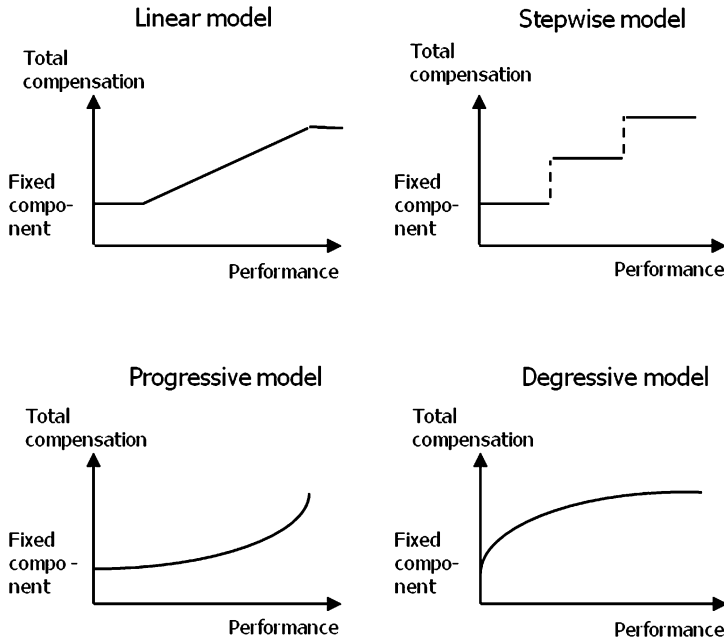


Fig. 3.18 Alternative performance-related pay models (Homburg and Krohmer 2006, p. 1266)

KPIs listed below, by contrast, have proven helpful for sales performance management in the software industry:

- Number of leads generated by direct marketing,
- Volume of addresses,
- Number of first visits,
- Opportunities weighted by closure probability,
- Number of solution presentations at customer,
- Number of proposals/quotations submitted,
- Quotation value,
- Number of contract negotiation meetings,
- Number of deal closures,
- Lost opportunities/unsuccessful quotations, and
- Customer visits per salesperson per month.

These KPIs can also be used for calculating variable, performance-based pay for sales staff. In line with principal-agent theory, the objective is to develop an incentive-compatible compensation model (see Sect. 2.5), i.e., design a model that motivates the agent (sales person) to pursue the goals of the organization (principal). A core feature of such a system is the division of pay into a fixed and a variable component. Figure 3.18 shows various basic alternatives.

The linear model is standard practice. The variable component should amount to about 30–40 % of the total target compensation. One of the key issues to be addressed is the selection of parameters used to determine the variable component.

In the software industry, sales targets or order inflow targets are most often used to determine the variable component. Concrete answers are needed to the following questions:

- What sales period will be used (usually the fiscal year)?
- What counts toward fulfillment of sales targets?

The second question can be highly contentious. The most delicate question for software companies is how maintenance business is handled. The significance of this question is apparent when we remember that maintenance income is generated over long periods and can account for up to 80 % of a software company's total annual sales (see [Sect. 1.5](#)).

During the sales process, setting the right price is a constant consideration, as it is crucial to a successful sales strategy. Against this background, the following section will investigate software providers' pricing strategies.

3.3 Pricing Strategies

3.3.1 Background

Pricing plays a key role in most organizations' strategies (Simon 1992, p. 7). It directly affects revenues and therefore, in the long term, profits. Incorrect decisions can jeopardize the company's reputation and customer relationships. Despite its importance, pricing strategies are often deficient in a number of respects, including lack of rationality in the shape of ad-hoc or arbitrary decisions (Florissen 2008, p. 85). Small and midsize enterprises are by no means the only ones to frequently rely on gut feeling when they make pricing decisions. But basing pricing strategies on empirical data can make a great deal of economic sense. Studies show that price adjustment at the right time usually has a greater impact on profit than a reduction in costs. For instance, a price adjustment of just 1 % can lead to a rise in operating profit of some 8 % (Marn et al. 2003).

However, conventional pricing models are not directly applicable to software products (Bontis and Chung 2000, p. 246). Of the many characteristics of software as a good (mentioned in [Sect. 2.1](#)), the fact that it can be duplicated virtually for nothing is particularly significant when it comes to setting a price. To recap: it is relatively expensive to develop a first copy of a digital good. But the marginal cost of an additional copy is near zero. How does this affect pricing? First of all, clearly, cost-based pricing must be ruled out. Demand- or value-based pricing makes far more sense. This means that software providers need to base their prices on how much their potential customers are willing to pay. Shapiro and Varian describe this relationship as follows: "cost-based pricing just doesn't work [...]."

You must price your information goods according to consumer value, not according to your production cost.” (Shapiro and Varian 1998, p. 3).

In principle, the cost structure of digital goods lends itself to low-price strategies. These can be useful when, for instance, a software provider wants to squeeze out an established provider on a network market. Since the variable costs of the software product are negligible, the contribution margin would not be negative, even if the software were given away. This does not apply to physical products, as they are subject to variable costs. We will see below that the cost structure of digital goods also lends itself to price bundling, for example.

It would be a mistake, however, to assume that in economic and, specifically, in pricing terms software can be treated like any other digital good. While doing so might make sense for a software vendor that generates all or most of its revenues through license sales, it is by no means the general rule. In fact, consulting and support services do incur costs.

The following section provides an overview of the parameters that can be used in pricing software products.

3.3.2 Pricing Models for Software Products

3.3.2.1 Overview

Software products may be offered in many different forms. Over time, pricing models for software have changed fundamentally. Whereas during the mainframe era, prices were usually based on computing power, pricing models based on user numbers (licensing models) have been prevalent in the recent past (Bontis and Chung 2000, pp. 247–248). Today, software providers are increasingly offering usage-based pricing models, a topic we will examine in greater depth in [Sect. 3.3.2.3](#).

Since there is no universally valid pricing model for software providers (Bontis and Chung 2000, p. 246) and pricing models can be comprised of multiple components, we will next describe various pricing parameters for software products. [Figure 3.19](#) shows an overview of the parameters we will be discussing (Lehmann and Buxmann 2009).

Software providers’ pricing models generally comprise a combination of parameters. The pricing model can include multiple sub-items from each column.

3.3.2.2 Price Formation

The provider defines how the price is to be formed. Both the pricing basis and the degree of customer interaction must be taken into consideration.

There are essentially three ways to determine prices (Homburg and Krohmer 2006, p. 720; Nieschlag et al. 2002, pp. 810–814):

- Based on cost,
- Based on value or demand, and
- Based on competition.

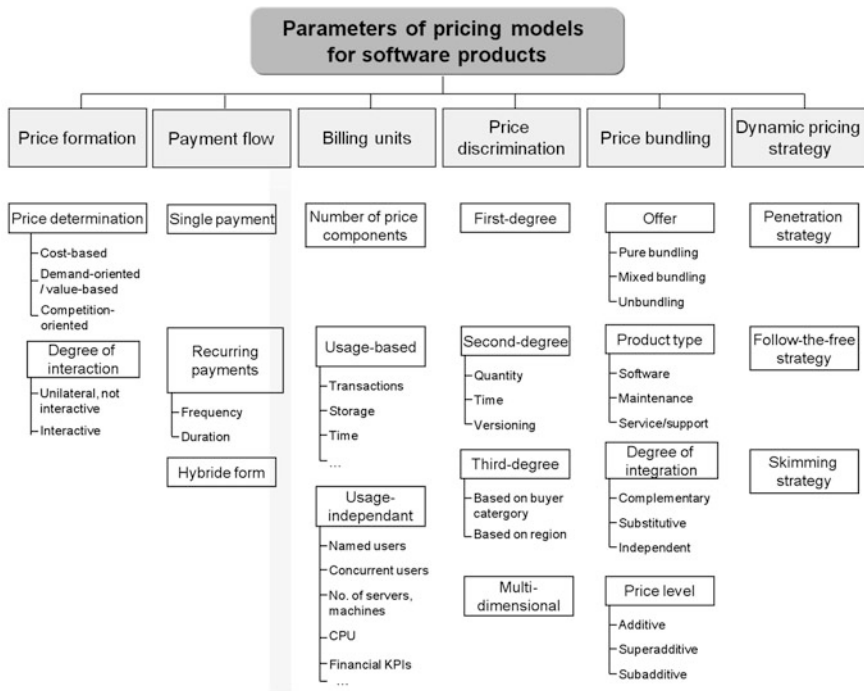


Fig. 3.19 Parameters of pricing models for software products

The cost-based approach determines the price using cost accounting methods (Diller 2008, pp. 310–311). However, this method of determining the price is of little significance when it comes to software licenses and digital goods in light of their unique cost structure. When pricing SaaS solutions, on the other hand, it can make sense to take costs into account.

Demand- or value-based pricing is oriented toward the level of demand for the product (Homburg and Krohmer 2006, pp. 720–721). Here, the significant factor is how much the customer values the product, rather than the product’s cost (Harmon et al. 2005, p. 1).

Competition-based pricing takes into account the prices offered by competitors and their price-related behavior (Homburg and Krohmer 2006, p. 747). The attractiveness of a competing product to a customer partly depends on the homogeneity of the products and the structure of the market (Nieschlag et al. 2002, p. 813). In the software industry, network effects and the resulting customer lock-in effects make it essential for software providers to gain a large market share, especially when their product is very similar to the competitor’s. Competition-based pricing therefore plays an important role for software products in addition to value-based pricing.

Another pricing parameter is the degree of interaction. Noninteractive price formation is when the provider sets the price unilaterally, without the customer

having any say in the matter. Interactive pricing, in contrast, requires interaction between customer and provider. Examples of interactive pricing include negotiations and (online) auctions (e.g., Schmidt et al. 1998). But generally speaking, auctions of digital goods, including software, make little economic sense (Shapiro and Varian 1999, p. 23).

3.3.2.3 Structure of Payment Flow

There are essentially two types of payment flows for software: Customers can make a one-time payment and acquire the right to use the software for an unlimited period, or they make regular, recurring payments. The two types can also be combined (Kittlaus et al. 2004, S. 82).

The single payment option corresponds to the software licensing model widely used today. By purchasing a license, the customer normally acquires unrestricted usage rights.

Recurring payments can vary in frequency and duration. For example, customers can agree to pay monthly or annual subscription rates over a two-year period in order to use the software. These pricing models are primarily used for SaaS solutions (Cusumano 2007, p. 20), where customers use the provider's software via the Internet for the agreed payment period (this is sometimes referred to as a subscription or rental model; Buxmann et al. 2008b). This kind of pricing model benefits users because it enables them to employ software cost-effectively even for brief periods, since normally the monthly payments are substantially lower than a one-time payment for licenses (Cusumano 2007, p. 20). However, this customer advantage makes higher financial demands on the provider. For example, SaaS providers often find it hard to break into the black (Hill 2008, p. 48). According to a survey by SIIA et al. (2006, p. 5), US-based software providers expect that the subscription model will become more common than one-time payments in the form of license purchases (see also Sect. 3.3.3).

Hybrid payment models that combine one-time and regular payments are another option. For example, it is common to purchase a software license in conjunction with a software maintenance agreement. These usually specify annual payments amounting to a certain percentage of the (one-time) license payment. At present, many software providers charge a maintenance percentage of around 20 %. The advantage of this model for providers is that it generates relatively uniform payment flows. In Sect. 3.3.3, we will provide some empirical findings regarding forms of payment for SaaS solutions.

3.3.2.4 Billing Units

The choice of billing units is another way in which pricing models can be shaped. In other words, the price can be set per user or based on a time factor, for example. The billing unit plays a key role in determining whether the customer feels the provider's pricing model is *fair*.

First the provider determines how many components the pricing model consists of (Skiera 1999b). Each component is based on a pricing unit. For instance, the

Table 3.4 Examples of usage-based billing units (Lehmann and Buxmann 2009)

Billing unit	Description
Transaction	The price depends on the number of transactions completed using the software. Both transactions in the technical sense (e.g., Web service calls) and in the business sense (e.g., number of delivery items processed) can be applied
Storage	The price is measured in units of storage capacity utilized (e.g., per GB)
Time	The price is determined by the actual duration of use of the software (e.g., per minute)

Table 3.5 Examples of usage-independent pricing (units) (Lehmann and Buxmann 2009)

Billing unit	Description
Named user	Software usage rights are linked to specific persons and pricing is based on particular individuals
Concurrent user	This option enables simultaneous use of software by a predefined number of users
Server/machine	Customers are charged per server or machine. Software usage rights are linked to a server or machine
CPU	The software price is calculated by the number of CPUs it runs on
Master data	Pricing depends on the amount of master data maintained (e.g., customers, suppliers, employees, inventory items, rental units, land parcels, managed assets)
Sites	Prices are calculated by site. This can include special types of site (e.g., mines)
Production volume	Pricing is based on a metric of production (e. g. barrels of oil per day)
Key performance indicators	Pricing is based on Key Performance Indicators (e. g. revenue, expenses, budget)

pricing model can be divided into a fixed monthly charge and a usage-based component, such as storage capacity utilized. Skiera (1999b) showed that service providers can increase their profits considerably by using two pricing components rather than one.

As we indicated in the above example, billing units can be either usage-based or unrelated to actual usage of the software. In principle, many different billing units are conceivable. They can also be industry-specific, such as the number of rental units managed in the case of property management software. Table 3.4 includes examples of usage-based billing units.

Implementing usage-based pricing can result in fixed and variable administration costs, for instance, for monitoring usage and for billing.

Usage-independent billing units are not dependent on to what extent the software is actually used. See examples in Table 3.5.

From the software provider's perspective, one benefit of usage-independent pricing is that customers are generally prepared to pay more for unlimited usage

(Sundararajan 2004, p. 1661). Many customers overestimate how much they use (flat rate bias; Lambrecht and Skiera 2006, p. 221). Empirical findings on the prevalence of usage-based and/or usage-independent pricing models in the SaaS field are shown in [Sect. 3.3.3](#).

Billing units are not only important for the customer, but also in terms of price discrimination. This will be discussed in more depth below.

3.3.2.5 Price Discrimination Strategies

Price discrimination means charging customers different prices for essentially the same product (e. g. Diller 2008, p. 227; Skiera and Spann 2000; Pepels 1998, p. 89). The goal of the provider is to capture more of the consumer surplus. In contrast to a pricing model with consistent prices, this can be achieved by figuring in varying degrees of consumer willingness to pay (WTP). Because different customers estimate the product's value differently, providers can differentiate their prices and achieve higher turnover (Diller 2008, p. 227). An example of this is an audio CD available in three separate versions with different price tags:

- The premium version contains a wide range of additional features, such as song booklets, a multimedia section enabling access to an exclusive Internet offering, or bonus tracks.
- The standard version offers the features of a conventional album.
- The basic version comprises simply the disk in a sleeve with no booklet.

Pigou (1929) distinguishes between first, second, and third degree price discrimination.

At first glance, the best strategy for software providers seems to be first *degree discrimination*. The idea behind this is to distinguish the price individually by offering the product exactly at the customer's reservation price (RP), i.e., the maximum amount the customer is willing to pay. Minimum price thresholds do not have to be considered in the case of digital goods because the variable costs are negligible. This type of pricing strategy is not a practical option for software providers in the business-to-consumer sector, of course, because they have millions of customers and it is impossible to make even a rough estimate of each customer's WTP. In contrast, vendors of standard business-to-business software are clearly perfecting the art of price discrimination. Prices in this segment are often complex and not very transparent. Price lists can be hundreds of pages long, and there is huge scope for negotiating contractual terms. This applies not only to software licenses but particularly to complementary consulting services. However, providers of standard software generally cannot afford too much in the way of price discrimination. For example, customers may well exchange notes in user groups and discover that others have paid a different price for the same software product.

Second degree price discrimination plays a major role in digital goods (Linde 2008, p. 209). It is based on the principle of self-selection, i.e., the customer selects its own product-price combination (Varian 1997, p. 193). Skiera (1999a, p. 287)

makes a distinction among quantity-, time- and performance-based price discrimination with self-selection.¹

In the case of quantity-based price discrimination, the average price per unit depends on the total amount purchased. Flat rate is included in this category too, since the average price per unit depends on the consumer's overall use (Skiera and Spann 2000). This type of volume discount is widespread for software licenses, especially for key accounts. Discounts in excess of 50 % are not uncommon.

Time-based price discrimination targets degrees of consumers' WTP at various times (Skiera and Spann 1998). An example of time-based price discrimination is not charging a fee to disclose stock market prices with a time delay, while charging for real-time prices. There are a host of other examples, such as seasonal pricing. One that applies to the software industry is pricing customer service according to the time of day.

Another type featuring self-selection is performance-based price discrimination. This is when relatively minor changes are made in service scope or quality (Diller 2008, p. 237) and the resulting product variants offered at different prices. In conjunction with product differentiation, this is frequently referred to as versioning (Varian 1997; Viswanathan and Anandalingam 2005).

Offering several versions of a product is seen as especially profitable for digital goods because of the cost structure (Viswanathan and Anandalingam 2005, p. 269). In the context of network markets, inexpensive variants can lead to greater market penetration. This shows that software products generally are well-suited to this type of price discrimination (Bhargava and Choudhary 2008, p. 1029). Software providers often develop a high-quality, feature-rich product to begin with, so they can then remove certain functionality and offer consumers different versions (Shapiro and Varian 1999, p. 63). Some examples of performance-based price discrimination include the Home Basic, Home Premium, Professional, and Ultimate versions of Microsoft Windows 7, which vary in functional scope and price.

It is important to bear in mind that too many different versions can be confusing for consumers and make more work for the provider (Viswanathan and Anandalingam 2005, p. 269). Because consumers exhibit extremeness aversion, the rule of thumb for information goods is to offer three versions so that the customer can compromise by selecting the mid-range variant (Varian 1997, p. 200; Simonson and Tversky 1992; Smith and Nagle 1995). The basic idea is that undecided consumers will tend to choose the product of medium quality. The following example illustrates this principle: If a fast food chain were to offer beverages in large and small sizes only, some customers with no clear preference would probably select the small size. But if the vendor also offered a jumbo size, and the new medium size were identical to what was previously the large size, many customers would choose the new medium size (Varian 1997, p. 199 f.).

Bhargava and Choudhary (2008) recommend that companies consider offering additional lower quality variants when variable costs are decreasing. The authors

¹ Because of its minor importance for the software industry, search-related price discrimination has been omitted.

give formal evidence that decreasing variable costs make versioning more profitable for providers because that way, they acquire additional customers with a lower WTP (Bhargava and Choudhary 2008, p. 1031).

Third degree price discrimination is based on how the provider segments the market (e.g., Diller 2008, p. 229). In contrast to second degree price discrimination, the consumer cannot self-select. There are two types: location-based and customer based (Skiera and Spann 2000).

The latter is often used to create a lock-in effect: “Although software producers don’t hang around outside of schoolyards pushing their products (yet), the motivation is much the same.” (Shapiro and Varian 1998, p. 46). For example, a software provider could give away its products to a specific group of consumers to achieve lock-in effects. The idea is that the pupils will learn to navigate the software and when they become paying consumers, will be more likely to buy the provider’s product.

A simple kind of location-based discrimination is to sell products in different locations at different prices. Software licenses are sometimes priced this way, as are associated service agreements and consulting contracts.

Price discrimination that includes more than one dimension is called *multidimensional* (Skiera and Spann 2002, p. 279), and is very common practice. For example, pricing can be based both on location and on quantity. The provider has different prices for every country or region as well as a pricing model dependent on the quantity purchased. Multidimensional price discrimination can achieve a finer customer segmentation, which enables providers to exploit customers’ existing WTP even more fully. But the complexity of the pricing model should be limited to avoid confusing the consumer, and to ensure that the provider’s billing process remains feasible (Skiera and Spann 2002, p. 279).

3.3.2.6 Price Bundling

Price bundling is another parameter relevant to software pricing. In general, it consists of packaging multiple distinct offerings (products, services, and/or rights) from one or more providers and selling the package at a single price (Diller 2008, p. 240). It is sometimes considered a special form of price discrimination (Skiera et al. 2005, p. 290; Diller 2008, p. 240). Because of its significance in the software industry, we will devote a complete section to price bundling in relation to software pricing.

A variety of goals can be achieved through bundling. First and foremost, it can be used as a means of price discrimination (see [Sect. 3.3.2.5](#)) (Viswanathan and Anandalingam 2005, p. 264). While conventional methods of discrimination require comparatively detailed knowledge of each product’s RP, this does not apply to bundling to the same extent (Adams and Yellen 1976, p. 476). Bakos and Brynjolfsson (1999) explain this with reference to the law of large numbers. According to this, it is simpler for the provider to estimate customers’ WTP for a bundle that includes multiple products than for each product individually. This is because the WTP distribution for the bundle shows fewer extreme values

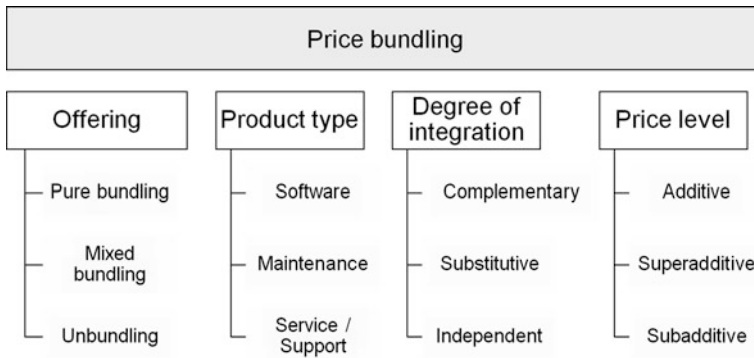


Fig. 3.20 Aspects of price bundling (Lehmann and Buxmann 2009)

(Viswanathan and Anandalingam 2005, p. 264). However, Wu et al. (2008, pp. 608–609) argue that this only applies when variable costs are zero. If there are any variable costs at all, even if they are extremely low, a bundle composed of numerous elements generates significant costs, which will lessen the potential advantages of bundling.

Because the software industry is heavily influenced by network effects, bundling can be advantageous to providers, as it drives the proliferation of (additional) products. For example, Adobe’s Creative Suite not only includes the software for photo editing and layout design, but also the software for creating PDF files. As a result, sales of the Creative Suite also promote the spread of PDF documents. Moreover, a bundling strategy may impede market entry for potential competitors (Nalebuff 2004), e.g., for providers that offer only one of the products in the bundle. It can also reduce billing and shipping costs, as multiple products are sold in a single transaction (Viswanathan and Anandalingam 2005, p. 264; Adams and Yellen 1976, pp. 475–476).

We will explain the aspects of price bundling shown in Fig. 3.20 before discussing the factors that determine how advantageous bundling strategies will be.

Software providers’ offerings can include pure and mixed bundling as well as unbundling. Pure bundling means that products are only offered as part of a package. If the customer can choose between purchasing the bundle or buying each product separately, this is referred to as mixed bundling. Unbundling is when the customer can only purchase the products separately (Adams and Yellen 1976; Schmalensee 1984, pp. 212, 475; Olderog and Skiera 2000, p. 140). Another option is customized bundling, in which the customer can choose, within specified limits, which products to include in the bundle. The provider determines merely the price and scope (Hitt and Chen 2005). Wu and Anandalingam (2002) show that offering multiple, customized bundles can be beneficial to a monopolistic provider of information goods. Assuming incomplete information, Wu et al. (2008) found that customized bundling is more profitable than unbundling.

The *product type* is another aspect of price bundling. The products offered together in the bundle can be very different in nature. In the software industry, the product types are usually the software itself, maintenance, support, and other services. Today, software vendors often generate revenue from three sources—licenses, maintenance, and other services—in equal measure (Cusumano 2007, p. 19). These three types of offering can be bundled in various ways.

Products in a bundle can also be described according to their *degree of integration*. Elements of a bundle can be complementary (Diller 2008, p. 241), substitutive, or independent of each other. Bakos and Brynjolfsson (1999) discovered that bundling a large number of unrelated information goods can be profitable. Their model also lends itself to analyzing complementary and substitutive elements of bundles. Stremersch and Tellis (2002) suggest that when the purpose of bundling is to add value for customers in comparison to using or consuming the products separately, it is better to use the term *product bundles* rather than price bundles.

Methods for determining bundle *pricing* can be additive, superadditive, or subadditive. In the first instance, the bundle price corresponds to the sum of the individual prices. Superadditive bundles are priced higher than the sum of the individual components' price tags, while subadditive bundles are priced lower than the sum (Diller 2008, pp. 240–241). The latter case, i.e., a bundle that offers a discount on individual prices, is considered the norm (Diller 2008, p. 241; Viswanathan and Anandalingam 2005, p. 264). A survey by Günther et al. (2007, p. 139) revealed that most respondents expected a lower overall price for the bundle when purchasing bundled Web services. For example, the price for the Microsoft Office bundle is significantly lower than the sum of the components' prices. When Microsoft distributed the Windows operating system bundled with Media Player, customers were under the impression that the latter product was thrown in for free. This type of bundling strategy can be used, among other things, to include new applications with older products in order to motivate customers to upgrade or opt for maintenance services (Cusumano 2007, p. 20). However, it is important to note that product bundling can contravene antitrust legislation. A prime example of this was the action taken by the European Commission against Microsoft for bundling its Windows operating system with Internet Explorer.

Olderog and Skiera (2000), among others, looked at which factors influence the benefits conferred by bundling, on the basis of a model by Schmalensee (1984). The success of bundling strategies depends primarily on two factors: first, the type and degree of correlation between the RPs, and second, the size of the variable costs compared to the RPs. These two factors will be discussed below. RPs are positively correlated when the consumers who would be prepared to pay a high (low) price for product A would also pay a high (low) price for product B. A negative correlation exists if the customers who would tend to pay a high (low) price for product A would be willing to pay a low (high) price for product B. In principle, the more negative the RP correlation, the more advantageous a bundling strategy will be for a provider. This is because a large negative correlation leads to lower variance in the bundle's RPs, creating a more homogeneous

Fig. 3.21 Example distributions of reservation prices for two products (Olderog and Skiera 2000, p. 143)

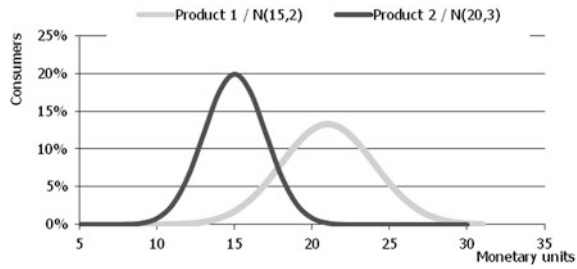
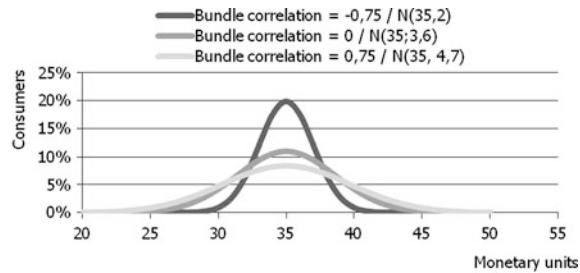


Fig. 3.22 Effect of RP correlations on the homogeneity of the RPs for the bundle (Olderog and Skiera 2000, p. 143)



demand structure (Olderog and Skiera 2000, p. 142). Figure 3.21 shows the RP distribution for two products and this relationship is depicted graphically.

Figure 3.22 shows how the correlation between the RPs for the individual products affects the homogeneity of the RPs for the bundle.

The homogeneity of the demand structure for the bundle increases (decreases), if the RP correlation is negative (positive). Figure 3.23 shows that the provider can increase the profit (dark area) when the demand structure is homogeneous.

As explained earlier, the success of a price bundling strategy also depends on the size of the variable costs in relation to consumer RPs (Olderog and Skiera 2000, p. 144; Bakos and Brynjolfsson 1999). We will now explain this using the simple numerical model shown in Table 3.6.

Here we are assuming that there are only two products, and comparing whether an unbundling or a bundling strategy is more successful when variable costs are relatively high. The first two lines give the maximum RPs of the two customers for products 1 and 2 and for the bundle comprising these two products. It is also clear that there is a negative correlation between the RPs of the two potential customers. The third line gives the optimum price for the two products and for the bundle. As the variable costs associated with making each of the two products are assumed to be seven monetary units, it is reasonable to set a price of eight monetary units for both products. This results in a positive profit contribution of one monetary unit each for products 1 and 2. Neglecting any fixed costs, the return generated by this bundling strategy is two monetary units. In light of this, it obviously makes no sense to pursue a bundling strategy as the variable costs are too high.

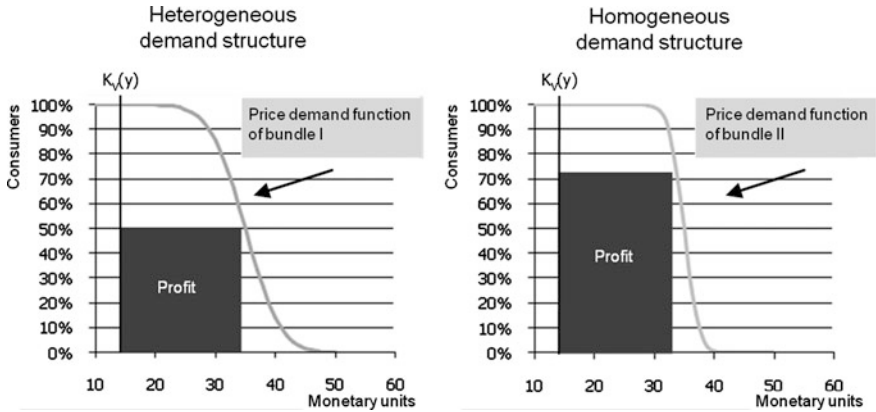


Fig. 3.23 Effect of a homogeneous demand structure on profit (Olderog and Skiera 2000, p. 144)

Table 3.6 Example of the effects of a bundling strategy when variable costs are relatively high

	Unbundling		Bundling
	Product 1	Product 2	Bundle
RP buyer $i = 1$	8	4	12
RP buyer $i = 2$	4	8	12
Variable costs	7	7	14
Optimal price	8	8	/
Profit contribution	1	1	/
Quantity sold	1	1	0
Profit	1	1	0
Total return	$2 >$		0

But how does the picture change when the variable costs are lower? To show this, we recalculate our simple example with variable costs of zero (see Table 3.7).

As we can see, due to the homogeneous demand structure, a bundling strategy allows the provider to exploit the consumers' RPs to the full. So with lower variable costs, the bundling strategy is more successful than the unbundling strategy.

Finally, Fig. 3.24 shows how the variable cost level affects the success of bundling strategies.

For our numerical example, we have shown the profits for the bundling or unbundling options in accordance with the variable cost level. It is apparent that the critical level for variable costs is four monetary units. If they exceed this amount, unbundling is the more profitable alternative. Conversely, when the

Table 3.7 Example for the effects of a bundling strategy for digital goods

	Unbundling		Bundling
	Product 1	Product 2	Bundle
RP buyer $i = 1$	8	4	12
RP buyer $i = 2$	4	8	12
Variable costs	0	0	0
Optimal price	8	8	12
Profit contribution	8	8	12
Quantity sold	1	1	2
Profit	8	8	24
Total return	16<		24

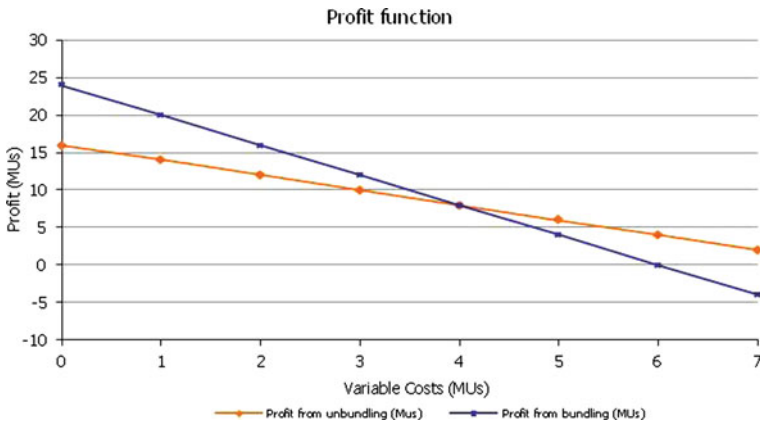


Fig. 3.24 Break-even point as a function of the variable costs

variable costs are lower than four monetary units, the provider gains more from bundling.

A further advantage of bundling for software providers is that it helps them to broaden the installed base of their products and thus generate network effects.

In conclusion, when there is a negative correlation between the RPs of the different products, bundling will tend to produce a more homogeneous demand structure. In addition, the lower the variable costs, the more advantageous bundling strategies become. For this reason, price bundling is a particularly useful

strategy for software products. It should also be noted that the ideal number of products in a bundle can also depend on the presence of constraints on customers' budgets (Bakos and Brynjolfsson 1999).

3.3.2.7 Dynamic Pricing Strategies

Unlike the approaches we have examined so far, dynamic pricing strategies are based on a multiperiod horizon in which time prices will usually change. The following sections will focus on penetration, follow-the-free, and skimming strategies. We will not discuss pulsation strategies (where vendors alternately raise and lower the prices over time) as we cannot imagine any useful applications of them in the software industry.

The purpose of employing a *penetration strategy* is to quickly acquire market share by means of low prices. A low-price strategy can play an important role in network effect markets in light of the startup problem and lock-in effects, which were mentioned earlier.

The presence of network effects means that the software industry lends itself to penetration strategies. Furthermore, the type of cost structure in this sector, featuring negligible variable costs, favors low-price strategies. In fact, even giving away software will not lead to negative profit margins. In some situations, it may even be necessary to make a loss in order to catch up with a competitor who has a head start. However, whether this kind of strategy makes sense depends heavily on the network effect factor (i.e., the size of the network effects in relation to the total utility). The higher the network effect factor, which we discussed in Sect. 2.2, the more vital low-price strategies will be (Buxmann 2002). Furthermore, a penetration strategy will be especially worthwhile, if the provider is offering a product that is incompatible with the market standard.

A penetration strategy can include a second stage in which the provider increases prices once a critical mass has been reached. Ahtiala (2006) showed experimentally that given the problem of software piracy, it benefits software providers initially to sell their products at a very reasonable cost to generate a lock-in effect, and only to offer subsequent upgrades at a higher price. To a certain extent, this strategy has been observed in the real world: some companies—including big names—have given away licenses for free in recent years.

The *follow-the-free* strategy comprises two stages: first the products are given away to generate lock-in, then revenue is generated through the sale of complementary products or premium versions to the existing customer base (Zerdick et al. 1999, pp. 191–194). For instance, a company could offer its software product for free, and charge fees for associated services such as installation, maintenance, user training, and customizing (Cusumano 2007, p. 21). An example from the software industry is the strategy of Adobe, which succeeded in making its PDF format the industry standard by this method.

The pricing strategy of Adobe Systems Inc.

Adobe Systems Inc. is a leading provider of graphics, design, publishing, image, and video processing software for Web and print production. In 1993, the company set new standards with the development of the document description standard, PDF (Portable Document Format).

Adobe's success is mainly down to its unique pricing strategy: offering its Acrobat Reader free for viewing PDF documents, and selling complementary PDF creation software. In 2009, Adobe posted more than 20 % of sales with its Acrobat Software (Knowledge Worker segment). Today, the free Acrobat Reader has an adoption rate of around 89 %. Figure 3.25 shows the breakdown of Adobe's sales per product segment.

Adobe Flash Player

In 2005, Adobe Systems acquired its competitor Macromedia Inc., originator of the now widespread Flash technology. With this technology, Adobe pursued a pricing strategy similar to the one employed for its Acrobat product line: the Flash Player used to view Flash files is available as free-ware. According to the company itself, the player is installed on about 98 % of Internet users' PCs worldwide. Customers who wish to create Flash components must purchase an authoring tool.

Through this complementary pricing strategy, i.e., distributing the viewing software free and selling the associated file creation programs, Adobe has succeeded in both cases to introduce a standard which is widely used and accepted, and to generate impressive sales.

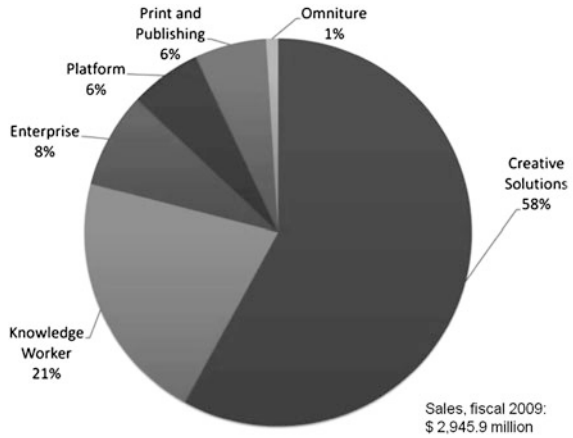
Adobe Systems Inc.

Adobe Systems Inc. was established in 1982 by John Warnock und Charles Geschke, the inventors of the PostScript document format. PostScript is a page description language used to describe the format of a printed page at the prepress stage. Building on PostScript's success, the company developed and launched the PDF standard in 1993. Since then, Adobe has published the source code. In January 2007, Adobe Systems submitted the PDF specification to the International Organization for Standardization (ISO), paving the way for it to become an official standard. The success of Adobe Systems Inc. is due in no small part to its innovative business strategy.

Sources www.adobe.com; Datamonitor (2006): Company Spotlight: Adobe Systems Incorporated, Financial Times Deutschland: http://www.ftd.de/technik/it_telekommunikation/133661.html.

In some cases, software products are offered on the basis of a *skimming strategy*, where the provider starts with a very high price and reduces it over time. The main purpose of this strategy is to exploit differences between consumers' RPs: in the high-price phase, consumers with a very high RP will acquire the

Fig. 3.25 Adobe's sales by product segment, 2009



product. Then, during the period in which the provider gradually reduces the price, it can exploit the RPs of the remaining consumers step by step. Computer games are a practical example of this strategy: very expensive when launched, they may even be distributed free of charge as magazine inserts at a later date.

3.3.3 Pricing Strategies of Software Providers: Empirical Findings

In this section, we will discuss how providers and users evaluate various pricing and licensing models. As few empirical findings on this subject have been published to date, most of the studies that exist include only a small number of parameters relating to software pricing.

Below we will discuss a study conducted in 2006 by the Software & Information Industry Association, which only addressed the parameter of billing units. We complement this with the results of expert interviews with users. The goal of our discussions is to cast more light on how users evaluate pricing, and why. However, it must be emphasized that the interviews are a qualitative rather than a representative sample of ERP system users.

The Software & Information Industry Association surveyed 698 experts. These comprised 487 software providers and 211 users (SIIA et al. 2006). Figure 3.26 gives an overview of the different licensing models, indicating which are preferred by software providers and users, respectively.

The most popular usage-independent billing unit among the users of software products is the number of concurrent users (see also Sect. 3.3.2.4.). Also attractive for users, but far less accepted, is licensing by servers or machines. Only a few customers favored billing by named users or processors. The least desirable form of billing is financial KPIs, preferred by only 1 % of users (SIIA et al. 2006, p. 7).

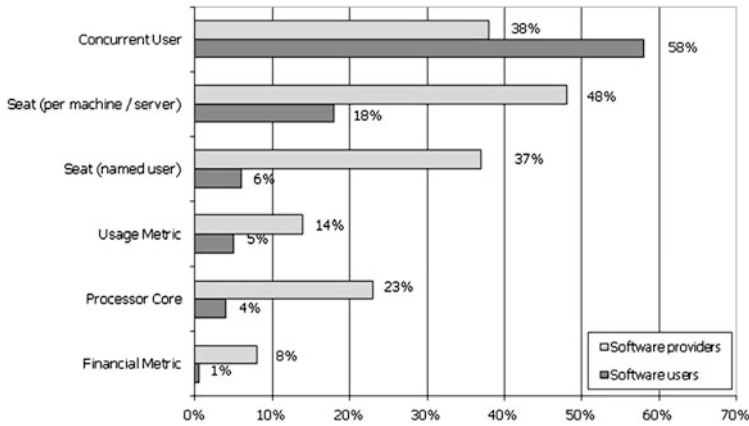


Fig. 3.26 The billing units preferred by software providers and users, respectively (SIIA et al. 2006, p. 7)

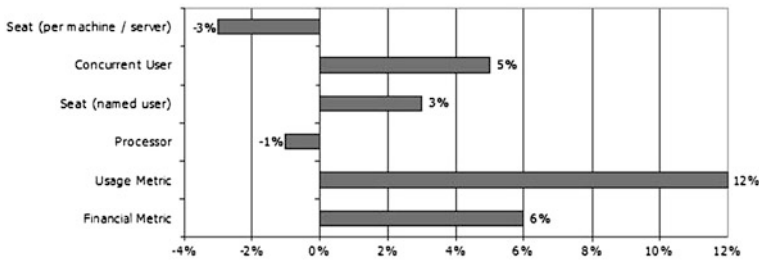


Fig. 3.27 How providers expect the use of billing units to change (SIIA et al. 2006, p. 7)

We will now look more closely at pricing models with usage-based billing units, as a key finding of the survey was that software providers expect this kind of model to become more widespread in future (see Fig. 3.27).

However, the findings also indicate that users are increasingly rejecting usage-based licensing models, as Fig. 3.28 illustrates

According to the SIIA survey, only one-fourth of customers were satisfied with the provider’s pricing and licensing strategies.

We will next present the findings of a survey of experts conducted in 2008 to discover the reasons why users evaluate certain pricing models as attractive or unattractive.

Only three out of ten experts interviewed regarded usage-based pricing models for software as an attractive alternative. Seven expressed their skepticism or rejection of this pricing model. The two main reasons they cited were the difficulty of predicting costs, and the possibility that costs would fluctuate significantly due

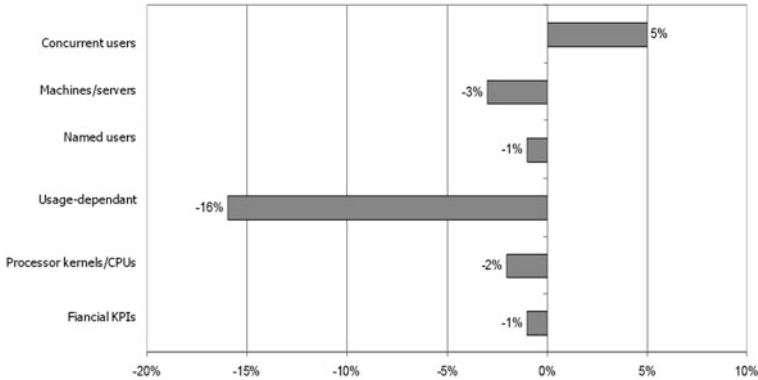


Fig. 3.28 Changes to users' billing unit preferences, 2005 compared with 2006 (SIIA et al. 2006, p. 7)

to varying usage levels. Difficulties lay not only in predicting costs, but also in determining the basis for billing. With a usage-based pricing model, it was essential that the effort needed to measure usage was not too high and that the model was straightforward and easy for users to understand. However, for some respondents, the structure of the model was consistent with their idea of fairness—these individuals declared themselves willing to pay more for using the software more (value-based pricing).

What survey participants did not like was the fact that most models force the user to pay for a certain level of usage in advance. Very often, additional fees are payable when actual usage exceeds the planned level. Conversely, users who do not fully exploit the planned usage level do not normally receive any refund or credit.

In general, respondents tended to look unfavorably at the pricing models currently used by software providers. This was due in part to the complexity of the models and the combination of multiple models. In this connection, respondents also expressed their dissatisfaction with IT outsourcing service providers. The latter often optimize their services on the basis of technical criteria, and this frequently leads to a mishmash of different licenses.

One respondent expressed his satisfaction with the conditions offered to large organizations. However, substantial discounts are only granted on licenses and not on maintenance fees. This has to do with accounting regulations. In the context of revenue recognition for multiple-element contracts (bundling), software vendors who prepare financial reports in accordance with US-GAAP are required to state a fair value, backed by objective evidence, for maintenance services yet to be provided under the terms of a contract. This is why these vendors do not offer discounts on maintenance services (Suermann 2006, pp. 112–114).

Asked what changes to pricing models user organizations would like to see, almost half of respondents expressed a wish for greater flexibility. They would like to be able to regularly adapt their agreements to changing user numbers, the option

of usage for limited periods (including periods with no usage), and new billing units that are independent of user numbers. However, flexible pricing models can also be a disadvantage to customers, as providers can exploit flexibility to their own ends. This is particularly likely to happen with business (ERP) software, since changing to a different provider would entail significant switching costs for the customer. It is, therefore, unclear who profits most from more flexible pricing models.

With regard to pricing of SaaS solutions, respondents suggested usage-based billing (four votes) and a flat fee (five votes). They believed that these models offer the option of limited usage periods and reducing entry barriers, and in the case of flat fees, better cost planning. In addition, users would like to see a much reduced notice period, although one CIO pointed out the risks to the user organization of short notice periods.

3.3.4 Approaches to Pricing for Custom Software Providers

The approaches discussed above apply predominantly to standard software providers. For example, product and price differentiation methods and dynamic pricing are obviously far less relevant to custom software providers. We will now look at how providers of custom software determine prices, both in terms of opportunities and constraints.

Setting or estimating lower price limits is a good starting point. For this purpose, traditional cost estimation methods for software development projects can be used, such as COCOMO (II) and the Function Point method (Balzert 2000).

It is reasonable to use methods like these because custom software projects are very difficult to plan in a precise, reliable way. Project budget and schedule overruns are the norm, not the exception.

Cost estimation methods gauge expected project effort and expense using certain parameters whose values are determined at the start of a development project. In its simplest form, COCOMO is based on an estimate of the number of lines of code. Project costs are estimated using tables based on empirical historical data. The Function Point method, by contrast—the method most commonly used to estimate the cost of software development projects—starts by evaluating project complexity in terms of parameters such as external inputs and outputs, user transactions, interfaces with external data sources, or the number of files used. Various inputs can be used to determine these quantities, such as specifications, entity-relationship diagrams, use case diagrams, screen layouts, and other documentation. Based on an evaluation of these criteria, tables are used to estimate the function points. A function point is an indicator of the scope and complexity of a proposed software solution.

Major providers of custom software employ methods such as these, whereas smaller-scale software companies are more likely to rely on instinct and experience. One advantage of the methods is that they are relatively easy to use. On the other

hand, they are also based ultimately on experience and empirical relationships, and some of their parameters are rather subjective. However, applying these methods can give providers valuable additional information for their decision making, particularly when it comes to major contracts. It often makes sense to apply these methods, given that this costs relatively little in comparison to the overall project costs. The methods deliver an estimate of project effort and expense, for example in terms of person-months, and of the likely development period.

The most significant source of expense for the software provider is the cost of the person-months invested. So, to determine a minimum price threshold for a development project, it makes sense to start by calculating the labor costs from the person-months. Labor includes both in-house employees and freelancers who are usually deployed during peak times. Labor costs for salaried employees comprise both fixed and variable pay components, employer contributions to social security, and training costs. Freelancers may be paid by hour or day, or may receive a flat rate for a given project or sub-project.

If the project effort expressed in person-months is weighted according to the costs of in-house versus freelance staff, this method will give a rough idea of the project costs, but no more than that. Manufacturers also use overhead costing and factor in indirect costs to determine the cost of making a given product. Although the same principle can be applied in the software industry, it is not generally necessary, because indirect costs—for offices, the use of software development environments, etc.—play a minor role in comparison to the labor costs.

Once these costs, with or without overhead costs, have been determined, the “only” thing left to do is to specify the mark-up. A large number of considerations must be brought into play, including the fact that the Function Point method tends to underestimate the actual costs.

The method described here is suitable for a cost-driven approach to pricing. Obviously, the prices determined this way will not necessarily be aligned with competitive demands or customers’ RPs. Against this background, other authors have suggested that it would be more appropriate to focus on the market situation when setting prices. However, this is extremely difficult to do when it comes to project business. After all, the unique, one-of-a-kind nature of projects makes it all but impossible to put a market price on them.

Psychological factors should also be taken into account. The relatively new discipline of behavioral pricing offers valuable insight here: it investigates potential customers’ attitudes to prices and pricing information, how they respond to prices offered, and how they use price information when evaluating products and making choices (Homburg and Koschate 2005). Some of its findings differ from the assumptions of classical price theory. For example, people often evaluate prices relative to a reference value, rather than in absolute terms: they often cannot recall prices; or they abandon a price search half-way through. Behavioral pricing primarily pursues a descriptive research approach and focuses mainly on cognitive processes not discussed by classical price theory.

Even though a cost-based pricing strategy has the drawbacks described above, it can still provide a good basis for determining whether a given project can be

performed in a cost-effective way, at least within a short-term framework. But it generally makes sense to base decisions on other objectives, too, such as opportunities for acquiring key customers, defending and extending market share or preventing competitors from penetrating the market.

Having devoted the preceding two sections to market-driven strategies, we will now turn our attention to development strategies.

3.4 Development Strategies

Academics and industry professionals have long concerned themselves with the efficient and effective development of software; previously, this was primarily considered in the context of user organizations, but is now increasingly analyzed from the perspective of software providers. A myriad of approaches have been proposed, tested, and discarded. Over the following pages, we will provide an overview of the most important approaches to have survived the test of time.

3.4.1 Structuring of the Software Development Process

The structuring of the development process is pivotal to the development of software. After a brief description of the early days of software development, we will discuss plan-based approaches. Agile development, which arose as a backlash to this approach, forms our next subject. Finally, we will compare the merits of the various approaches in conjunction with the relevant contextual parameters.

3.4.1.1 Ad-hoc Development

Along with the first program codes, one of the first software development methodologies also came into being in the 1950s and early 1960s. Under this methodology, also known as the “stagewise” model, software is developed in strictly sequential stages. In general, however, the development of software at this time more closely resembled a craft (hence the term “software crafting”), which depended on the abilities of the individual developers (Boehm 1986, p. 22; Dogs and Klimmer 2005, p. 15).

Software was generally developed according to the Code and Fix approach. This was not really a methodology as such. It simply meant developing functionality without much advance planning and continuing to make modifications to address any errors until the program ran without any error messages. The inevitable outcome of this process was highly unstructured and a hard-to-maintain program code: the infamous spaghetti code (Boehm 2006, p. 13).

3.4.1.2 Plan-Based Approach

The increasing popularity of computers also raised the demands on the associated software. As software became more and more unwieldy, cost, time, and quality

goals were often unachievable. As a result of this “software crisis”, the term “software engineering” was coined at a conference in 1968. Modeling software development more closely on the structured processes prevalent in engineering was intended to help overcome the software crisis (Dogs and Klimmer 2005, p. 16 ff.). Consequently, 1970 saw the invention of the “waterfall model” of software development. This built on the stagewise model mentioned above, and was regarded from then on as the archetype of plan-based development methodologies (Boehm 1986, p. 22).

The defining characteristic of plan-based methodologies is that software development is carried out in a series of standardized, usually sequential steps. Based on initial requirements elicitation and detailed planning, a range of artifacts is created in each stage, such as user requirements documents or technical designs. These form the basis for the next stages. After the actual programming, the software is tested for correct implementation.

However, in the ensuing years the waterfall model and plan-based approaches in general, were criticized. One criticism was that the sequential process made it difficult to accommodate changes to the customer requirements during the development process. This necessitated extensive, laborious modifications to the artifacts. The use of new methodologies, such as rapid prototyping, results in new approaches, including the spiral model (Boehm 1986, p. 21 ff.) and the V model (Hindel et al. 2004, p. 16), but none of this altered the fundamental limitations of the plan-based methodologies.

Apart from the term software engineering gaining widespread acceptance, none of the various approaches achieved the sought-after success. The old problems, cost blow-outs, lengthy project times, and poor quality, continued to exist. Numerous reasons have been put forward for this, including the following (Dogs and Klimmer 2005, p. 19 ff.):

- Plans quickly become out-of-date, as new customer requirements arise constantly.
- There are too many overheads (such as specifications and designs) which are irrelevant to the goal of developing high-quality software for the customer.
- Too little attention is paid to the human factor during the development process.
- The first tests take place too late on in the software development process.
- Not enough is learned from mistakes to benefit future projects.

In addition, increasing globalization and a more dynamic business environment are creating ever more volatile requirements. Critics allege that the inflexibility of plan-based approaches frequently means that software is developed which, by the time it is finished, the customer no longer needs.

3.4.1.3 Agile Approach

As a result of the dissatisfaction with plan-based approaches, a variety of “lightweight” approaches were developed in the early 1990s, independently of one another. The term “lightweight” is intended to convey a contrast with the

previously common “heavyweight” methodologies (in the sense of being process- and document-heavy) and enable a team to react as flexibly as possible to changes to the user requirements. Examples of this approach include Scrum, extreme Programming (XP), and Adaptive Software Development (ASD).

Lightweight methods follow a “just enough” approach. They attempt to eliminate processes that contribute minimal added value to the final software product. To this end, any artifacts that are created, such as specifications, are pared back to bare essentials, to keep the time and trouble of adjustment to a minimum in the event of changes. A key aim of lightweight approaches is to provide customers with the desired functionality incrementally (building on achievements to date)—as prioritized by the customer—in short, iterative development cycles. In addition, software tests do not simply occur at the end of projects, but are integrated into the development process.

Instead of seeing the constant changes to user requirements as a negative influence, lightweight methodologies see this as an opportunity to develop innovative software and so add value for customers. The role of developers is also different: Instead of extensive documentation, lightweight methodologies emphasize extensive collaboration among developers, including exchanging knowledge and experience. This means that the human factor plays a larger role than with plan-based approaches.

In February 2001, representatives of various lightweight methodologies met to establish commonalities and reach a joint understanding. One outcome was that participants agreed to use the term “agile” instead of lightweight, as the latter has negative connotations. In addition, they adopted the so-called “Agile Manifesto,” which was signed by all attendees and which sets forth their joint understanding with regard to agile methodologies (Fowler and Highsmith 2001, p. 28 ff.).

The values set out in the Agile Manifesto represent a shift in the importance placed on various aspects of the software development process. In the words of Fowler and Highsmith (2001, p. 29 ff.):

“We value:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.”

In addition to these four values, the Agile Manifesto also defines 12 principles around the values. But it deliberately avoids prescribing concrete programming practices. Rather, the principles are couched in sufficiently general terms, so that users can follow them yet still have sufficient creative freedom.

Critics allege that the principles underpinning agile development are neither new, nor do they represent a paradigm shift. In fact, the origins of iterative and incremental development can be traced back to the late 1960s and early 1970s (Fitzgerald et al. 2006, p. 200; Larman and Basili 2003, p. 48). What is new is the way the various agile methodologies interact, mutually reinforcing each other. In addition, proponents of agile methodologies recommend implementing

Table 3.8 Comparison of plan-based and agile software development approaches

Aspect	Plan-based	Agile
Attitude to change	Disturbance	Opportunity
Process	Sequential	Iterative
Artifacts	Comprehensive	As many as necessary
Role of developer	Resource	Decision maker
Releases	Full	Incremental

Table 3.9 Factors influencing the selection of methodology

Factor	Plan-based	Agile
User requirements	Stable	Unstable
Development team size	Tends to be large	Tends to be small
Security-critical systems	Suitable	Unsuitable
Type of software developed	Standard software	Custom software

iterations in a comprehensive way, to generate code and new software versions as fast as possible. Moreover, they advocate continuously modifying rough plans—providing a flexible way of dealing with problems and new requirements, which spring from the increasing speed of change in the business and technology worlds.

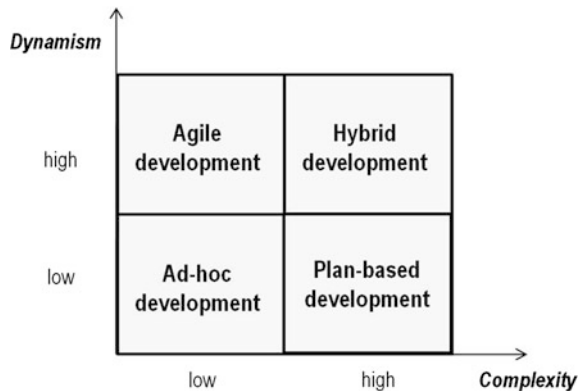
In any case, agile methodologies' basic principles are fundamentally at odds with those of plan-based software development, especially the advance preparation of plans in as much detail as possible, working in clearly identifiable phases and producing comprehensive documentation. The apparently irreconcilable differences between the two methodologies led to an increasing polarization of views, which was also known as the “Method War.” This episode involved agile advocates being labeled “hackers,” while supporters of plan-based approaches were likened to “dinosaurs” by their opponents. (Boehm and Turner 2003, p. xiii; Boehm 2002, p. 3; Beck and Boehm 2003, p. 45).

Table 3.8 provides an overview of the differences between plan-based and agile approaches in several key aspects.

However, the literature increasingly stresses that both agile and plan-based methodologies have their merits, with strengths in different areas. Neither methodology can be seen as fundamentally superior to the other (Boehm and Turner, pp. 148ff.). Table 3.9 shows which methodology appears to be best suited for which task in which application, on the basis of selected variables.

More recent approaches combine both plan-based and agile elements to exploit the strengths of both methodologies. For example, these hybrid methodologies plan the interaction of individual software components in great detail. The development of the components themselves, however, follows an agile approach.

Fig. 3.29 Advantages of the different development methodologies



Building on the issues discussed in this section, Fig. 3.29 illustrates which development methodology is generally best suited in which context, in the form of a dynamism and complexity matrix. The *dynamism* axis represents an environment with increasing economic or technological uncertainty, in which user requirements frequently change. *Complexity* includes aspects such as the number of software developers respectively, the size of the team involved or the need to integrate software and hardware from third parties.

Of course, this depiction is only a starting point for selecting an approach, as other factors also play an important role within a given organization, such as experience with a methodology or the specific type of the software being developed.

3.4.2 Software-Supported Software Development

The first attempts to generate software code automatically, based on models, took place in the 1980 and 1990. This was known as Computer Aided Software Engineering (CASE), and in retrospect, was only moderately successful. CASE approaches had a fundamental weakness (Schmidt 2006): their closed nature. CASE tools generate code for a single specific target environment. Moreover, the approach requires that all stages of software development are supported by just one tool, and executed sequentially in the traditional way.

New approaches, Model-Driven Engineering (MDE), address these two weak points. We are deliberately using this term to stress the development process (engineering). In the literature, the term Model-Driven Architecture is frequently used, which shifts the focus onto the architecture of the software creation tools. Essentially, MDE is based on models and transformation processes (Petrasch and Meimberg 2006):

- Models are used to represent application domains. There are two types, Platform Independent Models (PIMs) and Platform Specific Models (PSMs).

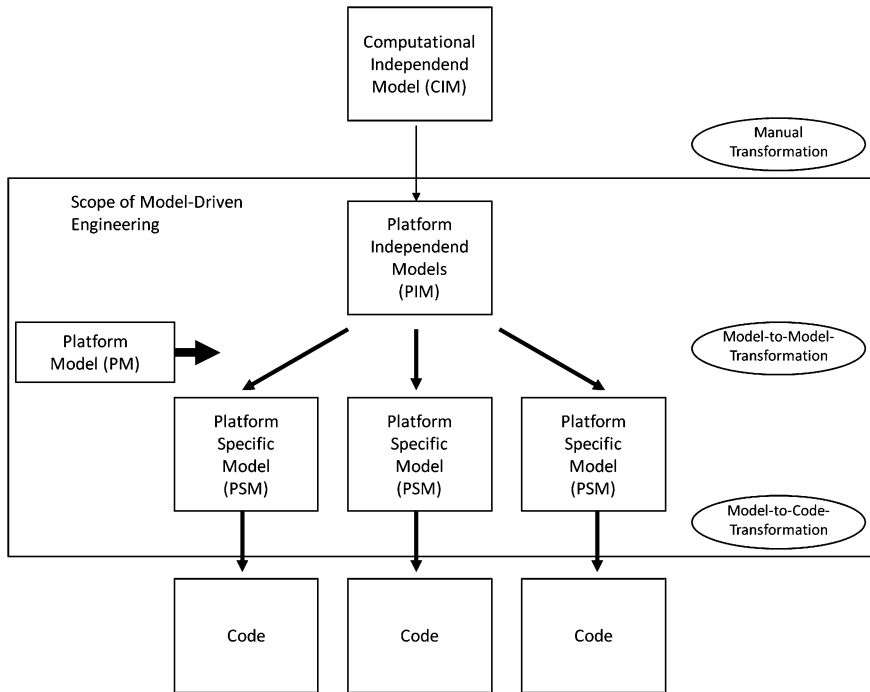


Fig. 3.30 Interaction of key concepts of Model-Driven-Engineering

- Transformation processes describe the automated mapping of language constructs of PIMs to PSMs (model-to-model transformation) and of PSMs to executable code (Model-to-code transformation).

Figure 3.30 shows how these two central MDE concepts interact, as well as the Computation Independent Model (CIM), which describes the domain to be supported by the software in informal, non-technical terms, and serves as a basis for the PIM.

Tools to support MDE typically include model editors, transformation editors, and tools and a repository. XML-based interfaces are available that allow different tools to exchange models.

The implementation of MDE approaches is still in its infancy. With respect to development costs, it can be assumed that the introduction and optimization of MDE approaches will give rise to non-project-related costs, while project-related costs in the later stages of development will be reduced. In other words: Implementing MDE only makes sense from the cost perspective, if

- the costs saved in development are not offset by additional costs for more extensive modeling and for the MDE implementation itself; and moreover
- a critical mass of projects is achieved.

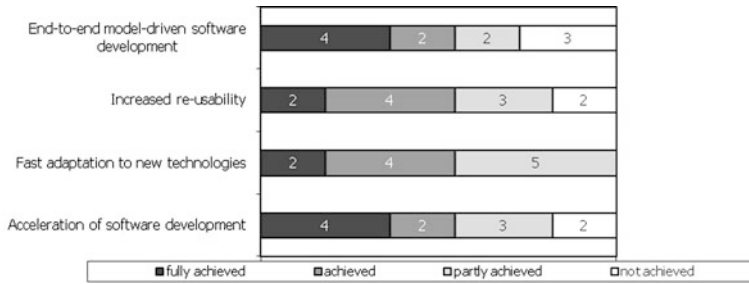


Fig. 3.31 Reported effects of MDE

MDE is a management concept that is yet to prove its effectiveness in the software industry. For this reason, we sent a questionnaire to 240 software providers in Germany asking for their view of selected aspects of this industrialization concept. 25 replied (Hess et al. 2007). This sample is small and not representative of the industry as a whole. However, it allows some initial conclusions to be drawn, which are presented below.

A significant majority of respondents were familiar with the concept of MDE, although it was more widely known in larger enterprises (annual net sales over 10 million euros) than in smaller companies. Almost half of the respondents already had some experiences with the MDE concept. These experiences varied considerably, as Fig. 3.31 shows. Nevertheless, there are some companies that successfully deployed the concept. In particular, they reported advantages in relation to end-to-end model-driven and accelerated software development.

More than half of the respondents expect MDE to play an important role in future. This assessment was more or less shared by the companies that had some experience of MDE. In summary, this suggests that MDE is a means for supporting software development that is seen by businesses in a positive light, but without euphoria.

3.4.3 HR Management in Software Development

Not least with the rise of agile development methodologies (see Sect. 3.4.1), it is increasingly accepted that the human factor plays a critical role in the software development process. The influence of such “soft” factors can play a greater role than technical factors as the type of technology or the use of a particular tool. The People Capability Maturity Model (CMM), developed by the Software Engineering Institute, addresses this issue. It applies the CMMI maturity model for assessing development processes to a company’s HR management (SEI 2010).

Having noted the importance of effective HR management for the success of software development projects, the following section looks in greater detail into the question of whether software developers have special characteristics that set

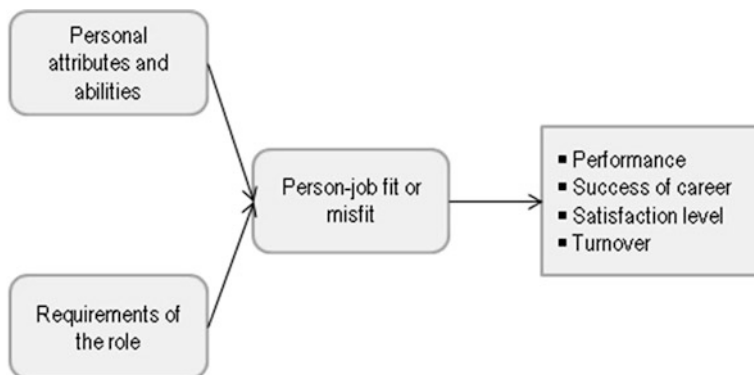


Fig. 3.32 Person-job fit (based on Lauver and Kristof-Brown 2001)

them apart from other professionals. Further to this, we will look at the “person-job fit” approach as a possible method to assign employees to roles and responsibilities. Finally, we will discuss the issue of employee motivation.

The ability to determine and even measure personality traits and types has caused increasing scientific interest in personality theories in the last few decades. Given that a person’s attitudes, beliefs, perceptions, and behavior are all influenced to a certain degree by their personality, the personality structure has a considerable impact on software development processes.

“Personality” refers to an individual’s unique psychological attributes, which, in turn, shape a multitude of characteristic behavior patterns. Consequently, personality traits can at least partially explain people’s interest in particular careers (Costa et al. 1984). Software developers generally possess a different personality type to that of the average population. They tend to be less dominated by their emotions, are more likely to be introverted and in many cases would rather work alone than as a team member (Capretz 2003).

Despite these similarities between software developers, there are of course considerable differences within this group; it would be wrong to talk of a “typical” software developer. Moreover, a software development project incorporates a variety of roles (e.g. team leader, quality manager, tester, programmer etc.), which have different demands.

The “person-job fit” approach focuses on assigning people to particular roles based on their specific abilities. The model, illustrated in Fig. 3.32, posits that an individual’s personal attributes and skills need to correspond with the requirements of their job. A closer fit generally leads to employees performing better, enjoying greater career success, and experiencing higher satisfaction levels, as well as less staff turnover.

Applying this to the software industry, Acuña et al. (2006) identify certain capabilities that are typical for people with particular personality traits. In the second step, these capabilities are matched against the specific requirements of eight different role profiles found in a software development project (see Fig. 3.33).

Software roles	Capabilities																			
	Intrapersonal						Organizational				Interpersonal			Management						
	Analysis	Decision-making	Independence	Innovation and creativity	Judgment	Tenacity	Stress tolerance	Self organization	Risk management	Environmental knowledge	Discipline	Environmental orientation	Customer service	Negotiation skills	Empathy	Sociability	Teamwork and cooperation	Coworker evaluation	Group leadership	Planning and organization
Team leader	✓	✓		✓			✓		✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
Quality manager	✓	✓	✓	✓			✓	✓				✓			✓		✓	✓	✓	✓
Requirements engineer	✓				✓		✓					✓	✓		✓	✓	✓			
Designer	✓	✓	✓			✓	✓				✓	✓			✓	✓	✓			
Programmer	✓	✓	✓			✓	✓	✓			✓	✓			✓	✓	✓			
Maintenance and support specialist						✓	✓	✓			✓	✓	✓		✓	✓	✓			
Tester			✓	✓		✓	✓	✓			✓	✓			✓	✓	✓			
Configuration manager			✓	✓		✓	✓	✓			✓	✓			✓	✓	✓			

Fig. 3.33 Matching developer roles with requirement profiles (Acuña et al. 2006, p. 98)

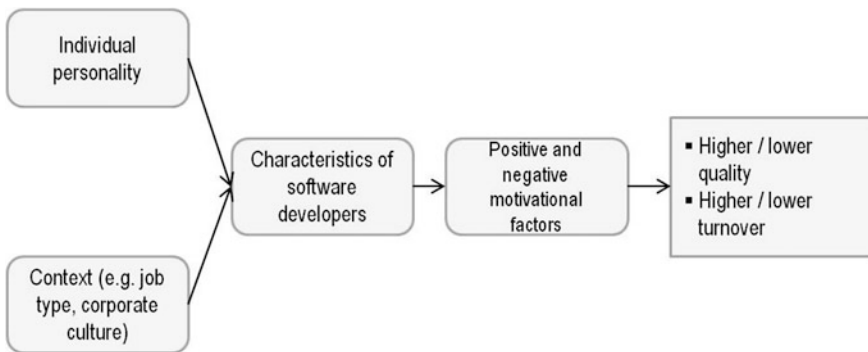


Fig. 3.34 Motivation of software developers (own illustration, based on Beecham et al. 2008)

Personality tests may also be used to determine whether an employee is suitable for a particular role. The capabilities identified in the test are then matched up with the role profile. It should be noted that this matching process provides only initial guidance as to whether an employee is suitable for a role—this model cannot take into account employees’ specific skills or experience, the particular requirements of the position, or organizational details.

Closely related to the assignment of people to roles and responsibilities is the issue of how best to motivate software developers. However, it is difficult to provide a general answer, as positive and negative motivational factors depend on the specific context and the person’s individual needs. Figure 3.34 offers a general model for explaining software developers’ motivation.

Although there are many unique, context-specific factors, the literature emphasizes identification with the task as the primary motivator for software developers: in addition to personal interest in the task, developers are motivated by

clearly defined targets and an appreciation of the significance of the task for the project as a whole. Other incentives include the existence of clear career paths and a varied and challenging range of activities (cf. Beecham et al. 2008).

A poor working environment, for example, lacking in key resources such as hardware and software, is commonly cited as a negative motivator. Others include poor management, such as calling superfluous meetings, and poor pay (cf. Beecham et al. 2008).

When implementing measures to increase motivation, attention should be paid to the previously discussed fit between the task and the person in question, as a misfit is a negative motivation. In other words, assigning the “wrong” person to the job can only be corrected to a limited extent by other means.

Part II

Specific Issues

4.1 Overview

There are few industries more international than the software industry. This can be traced back to the characteristics of software itself: code can be developed anywhere and delivered over the Internet in seconds. In contrast to a physical supply chain, the costs for transporting products or components are practically zero. This means providers can develop software in globally distributed projects. In turn, this keeps labor costs down, while also giving companies access to a worldwide talent pool. User companies that outsource their development tasks to software providers can also exploit these advantages.

But the globalization of the software industry is not only significant in terms of procurement and labor markets. Another key factor is that software can be sold very simply over the Internet. So it is not surprising that in terms of their organizational structures, all major providers are global in nature. The extent of the internationalization of the software industry can be seen by its almost complete lack of a home country advantage, in contrast to most other industries, where companies often enjoy considerably higher sales in their countries of origin than their foreign competitors.

Another form of competition takes place between locations. This is not only the case in Asia, where multiple software development centers are vying for supremacy, but also in Europe and the USA. Software companies' most valuable resources are their employees; the products they create are purely digital. That also has the effect that, "no company is as easy to relocate as a software company," in the words of Dietmar Hopp, one of the co-founders of SAP AG, commenting on the debate surrounding the establishment of a works council to represent employees at SAP.

In his multiple award-winning book, *The World is Flat*, journalist Thomas Friedman describes the internationalization of markets and identifies what he calls "flatteners", factors that are driving globalization (Friedman 2005). Against this background, we want to explore offshore software development and the resulting challenges and opportunities.

First, we will establish some basic definitions and introduce some varieties of outsourcing and offshoring. That leads us to a consideration of various organizational structures and an analysis of some of the issues related to IT service providers' choice of location. Finally, we discuss possible drivers and motives for outsourcing and offshoring, which provide a basis for assessing the success factors for these projects. Here, we focus on an investigation of the significance that geographical distance between customer and provider has for the success of a project. Our conclusions are partly based on our large-scale empirical study, which involved almost 500 CIOs. In addition, we also conducted expert interviews with leading employees from the following 10 software companies:

- Accenture,
- CGI,
- Cognizant,
- Covansys,
- Gamaxcom,
- HCL Technologies,
- Infosys Technologies,
- Starsoft Development Labs,
- Tatvasoft, and
- Wipro Technologies.

The empirical studies were complemented by 23 expert interviews.

4.2 Forms of Outsourcing and Offshoring

We will start this section by defining some basic terms: *Outsourcing* means contracting tasks or services, in our context, IT services in particular, to an external service provider (Mertens et al. 2005). If this provider is located in another country, this is known as *offshoring*; if not, the term *onshoring* is used. If the other party is an affiliated company in another country, we refer to *captive offshoring*. These terms are set out in Fig. 4.1.

The decision whether to outsource certain activities, such as software development, to third parties boils down to the classic “make or buy” decision. Transaction cost theory can be used to find a solution. We discussed this in detail in Sect. 2.4 .

Figure 4.1 makes a simple distinction between offshoring and captive offshoring. However, there is a sliding scale between these two forms of organization in terms of the intensity of the relationship, as shown in Fig. 4.2.

		Contractor is located...	
		In the same country	In another country
Internal activities are contracted to...	An affiliated company	–	Captive Offshoring
	An external company	Onshoring and Outsourcing	Offshoring and Outsourcing

Fig. 4.1 Outsourcing and offshoring (based on Mertens et al. 2005, p. 2)

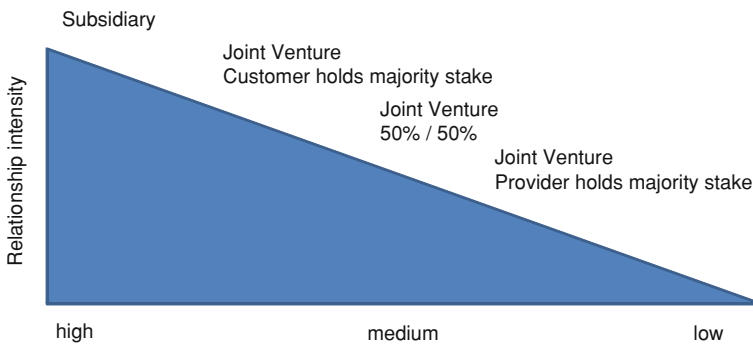


Fig. 4.2 Intensity of the relationship in various forms of organization (based on Bräutigam and Grabbe 2004, p. 179)

Companies can reap a host of benefits from the establishment of an offshore *subsidiary*. In addition to the primary goals of offshoring, cutting costs and access to a highly skilled workforce, captive offshoring can also help to penetrate new markets. Compared to outsourcing to an external company, captive offshoring generally permits a high level of control, not only over processes, but also over the company’s knowledge assets (Gadatsch 2006, p. 49).

However, captive offshoring also poses a number of challenges and risks that do not arise with external offshoring arrangements. Above all, this includes the cost of setting up a new business location (Willcocks and Lacity 2006, p. 6).

For this reason, many companies opt for a joint venture, reducing risk by entering into a partnership with a local company (Vashistha and Vashistha 2006, p. 172 ff.; Morstead and Blount 2003, p. 91). Players in many industries have successfully taken this route to international expansion. The following case study, about Software AG, shows how this is done and illustrates its potential.

SAG India

Software AG and iGate Global Solutions provide a good example of a joint venture. Together, these two companies set up a development and service center in India under the name Software AG India (SAG India). The company is headquartered in Pune, near Bombay/Mumbai.

A key advantage of the new joint venture was that SAG India could tap into a large pool of qualified IT professionals from the outset, thanks to iGate. This meant that it could become operational very quickly.

The new enterprise is incorporated as a Private Limited Company. Software AG holds 51 % of the shares, while iGate GS has 49 %. Each parent company is represented by two senior executives.

Initially, Software AG had only two employees in India: one employee who was responsible for legal and business matters (such as the formal establishment of SAG India), and another who was very familiar with the tasks that were to be carried out in India, and who, jointly with iGate employees, oversaw the hiring of new staff. The latter employee had already gained experience from Software AG's previous outsourcing activities and was therefore familiar with the culture in India. In retrospect, this contributed significantly to the venture's success.

Teaming up with iGate fast-tracked the process of establishing an off-shore site, as iGate already had the necessary premises, infrastructure, telephone systems, and so on. A further advantage of the joint venture was that iGate possessed many key contacts, so that, for example, Software AG did not need to find out for itself which government agency was responsible for which task.

Another form of organization that is closely related to the joint venture is the build-operate-transfer (BOT) model. In this model, as with a joint venture, the customer looks for an offshore partner, who then sets up and operates an offshore center. Only after the successful completion of these stages is the offshore center transferred to the customer's ownership (Vashistha and Vashistha 2006, p. 92). The advantage of this model is that some of the initial risks are at least partially borne by the partner company, while ultimately allowing the customer to have its own subsidiary. The form of organization with the least intensive relationship is outsourcing to external companies, shown on the far right in Fig. 4.2. In this form of offshoring, the customer devolves a considerable amount of control to the offshoring provider, which performs its services on a contractual basis. The advantage for the customer is that operational risks are relatively low. In addition, this form of organization can be implemented much more quickly than establishing a joint venture or subsidiary (Robinson and Kalakota 2005, p. 29).

Offshoring activities may also be categorized as nearshoring or farshoring. This distinction relates to the distance between customer and provider. As the names already indicate, *nearshoring* describes the outsourcing of IT services to a nearby

location, while *farshoring* denotes outsourcing to a distant one. This refers not only to geographical distance, but also cultural distance, and/or the number of time zones between the countries. A swarm of related buzzwords, such as *rightshoring* and *bestshoring*, have since been coined. We need not discuss them here.

From the perspective of German companies, offshoring to Eastern European countries or Ireland is commonly described as nearshoring. For their American counterparts, preferred nearshore locations include Mexico and Canada. Although a customer in California and a provider in Canada, for example in Halifax on the east coast, may be far apart in geographical terms, their relationship is still referred to as nearshoring. This is because there are few cultural and linguistic barriers. Typical farshoring destinations, from both an American and European perspective, include India, China, and Vietnam.

Before we turn to an overview of the outsourcing and offshoring markets, we should make it clear who the customers for these services actually are. Until now, the literature has tended to assume that these are companies whose core competencies do not include software development. However, software providers also outsource development tasks to offshore locations. In most cases, this entails captive offshoring or setting up a joint venture, as in the example of Software AG India. But this is by no means a rule. There are also examples of software companies outsourcing the development of entire modules to third-party providers. In summary, customers can be both users and providers of software.

4.3 Motives for Outsourcing and Offshoring

In this section, we will explore the primary motives behind offshoring. In particular, these include (Amberg and Wiener 2006, Hirschheim et al. 2006):

- Cost savings,
- Greater flexibility,
- Concentration on core competencies,
- Acquisition of knowledge and skills and
- Exploitation of the “follow-the-sun” principle.

4.3.1 Cost Savings

One of the most commonly cited reasons for outsourcing and/or offshoring is the expectation of reducing overheads, primarily through lower labor costs in offshore locations. The actual magnitude of these savings, however, is the subject of some debate. On one hand, the salaries of IT project leaders in offshore locations are often a fraction of those in the USA or Western Europe (see Fig. 4.3).

On the other hand, offshoring creates additional costs, especially resulting from greater coordination effort. A number of studies have been carried out, for example by Deutsche Bank Research and the outsourcing consultancy neoIT, to estimate

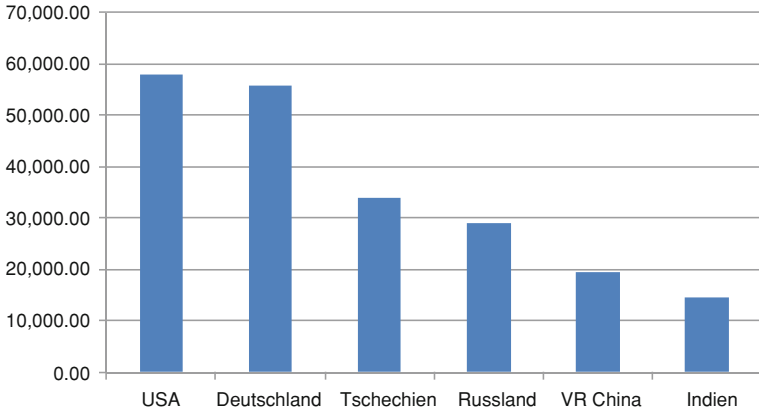


Fig. 4.3 Comparison of the annual salaries in euros of IT project managers (5–9 years experience) (Payscale 2010; own calculations)

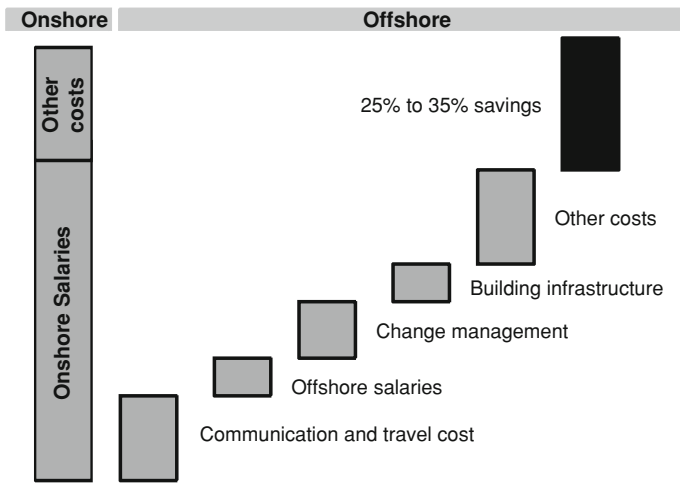


Fig. 4.4 Possible cost savings (Kublanov et al. 2005, p. 3)

the actual cost savings delivered by offshore projects. These results should be treated as rough indicators rather than precise, universally applicable figures. In the scenario of a software project executed offshore, neoIT regards the cost savings presented in Fig. 4.4 as achievable. This reveals that cost savings from considerably lower offshore salaries are only partially outweighed by higher expenditure, making savings of 25–35 % possible.

4.3.2 Greater Flexibility

For customers, increasing flexibility is another incentive for offshoring. In principle, this is achieved by buying the required services and skills only as needed. This has the advantage of converting fixed costs into variable ones.

4.3.3 Concentration on Core Competencies

In the face of increasing competition, many companies are choosing to focus on their core competencies. According to this school of thought, activities that fall outside the scope of these competencies should be outsourced (and, as the case may be, offshored). Assuming that highly specialized offshore providers can perform IT services more efficiently and effectively, the obvious conclusion is to outsource these tasks. The resources freed up as a result can then be devoted to strengthening core competencies. As for the offshore providers, the high volume of similar tasks enables them to exploit significant economies of scale, and therefore, at least in principle, offer their services at a more attractive price.

This oft-repeated argument initially seems straightforward and eminently plausible. On closer inspection, however, a number of complications come to light. What exactly are a bank's core competencies, when tasks such as credit checks are already among those being outsourced? Another example is the outsourcing of R&D in the automotive industry to suppliers. And in the software industry, we know of several cases where companies have outsourced part of their development activities to third parties—albeit with mixed results.

4.3.4 Acquisition of Knowledge and Skills

Another factor frequently cited as an incentive for offshoring relates to the employment market in offshore locations. For one thing, the number of available skilled workers is considerably higher than in organizations' home countries. And for another, these workers are highly motivated, even when it comes to simpler and less challenging tasks (Hirschheim et al. 2006, p. 6 ff.). In particular, India, the classic offshore location, but also China has much higher numbers of graduates than Western European countries (NASSCOM 2007).

In addition, the IT industry enjoys such a high status in India that many highly motivated Indian employees are keen to pursue a career in this arena (Vashistha and Vashistha 2006, p. 62).

But India, like many other Eastern European and Asian countries, not only produces graduates in large numbers. They are also very well trained. Indian IT professionals in particular are well versed in fundamental mathematical principles and programming (programming languages, knowledge of databases, etc.). Both the head of SAP's India-based development center and Daimler managers confirmed that the results and productivity of Indian, German, and American software

developers are much the same (Boes and Schwemmler 2005, p. 30). Graduates' very high skill levels are also confirmed by recent university rankings.

Against this background, it should come as no surprise that an online survey conducted by *Software* magazine found that 48 of the USA's top 86 software companies operate an R&D center in India (Aspray et al. 2006, S. 115).

However, we also need to caution against taking too rosy a view: For example, many software companies reported that their Indian workforces had relatively little loyalty to their employers. Consequently, it is not uncommon for employees to be lured away by competitors at very short notice. This has had the result that some software providers have several back-up members on every project team, in case some defect to a competitor halfway through.

4.3.5 Exploitation of the "Follow-the-Sun" Principle

The time difference between the customer's site and the offshore location provides an opportunity to increase the number of working hours in a day. In the ideal scenario, the on-site team can delegate a series of tasks to the offshore team before they leave. They are then presented with the completed tasks when they return to work the next morning (Hirschheim et al. 2006, p. 6; Amberg and Wiener 2006, p. 48; Thondavadi and Albert 2004, p. 18). On the other hand, time differences can also make it difficult to communicate in real time. We will study the opportunities and drawbacks more closely in [Sect. 4.6.3](#).

Last but not least, we want to return to the empirical study mentioned earlier. This revealed some interesting results relating to organizations' experiences with near- and farshoring. In general, companies with experience of offshoring evaluated the benefits of offshoring more favorably than those who had no such experience. Respondents were asked to evaluate some of the advantages and disadvantages on a scale from "not applicable" to "highly applicable". [Figure 4.5](#) shows the results obtained in relation to nearshoring. We recorded similar findings for farshoring.

These experiences are clearly shaping future plans: Almost half of the respondents' companies intend to outsource more work in the next few years. Smaller companies tend to be more cautious: Only 20 % of the sample reported plans to step up their offshoring activities.

4.4 Selection of Locations By Software Providers

As we saw in the previous section, companies' primary aim when outsourcing is to cut costs. Against this background, selecting the right location is strategically important for IT service providers. While the reduced labor costs in low-wage countries allow them to produce their services more cheaply, their choice of location should also take into account the benefits of being close to the customer. For example, SAP has outsourced the development of various modules to India

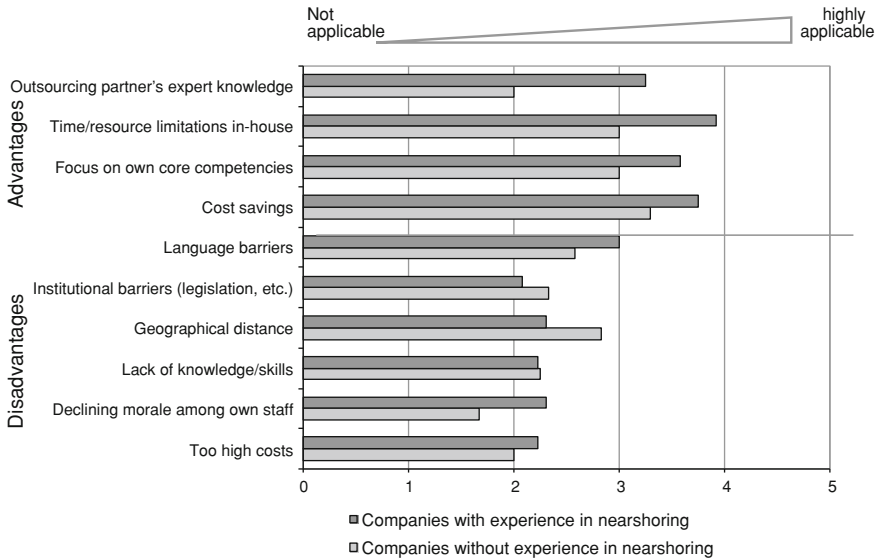


Fig. 4.5 Evaluation of the advantages and disadvantages of nearshoring

and intends to expand its activities on the subcontinent. This achieves its goal of cutting development costs. But this choice of location also offers better access to India, a high-potential market that ranks among the top eight in the world, according to SAP. Moreover, SAP has set up a network of fallback centers in several of Asia’s major cities (Mertens et al. 2005, p. 15 ff. and 21).

To date, providers have overwhelmingly shifted work from high-wage to low-wage countries. Recently, however, there has been some movement in the opposite direction. For example, Indian software companies are beginning to recruit young people from European and American universities and establish branch offices in high-wage countries. These providers are clearly willing to accept higher labor costs in order to be closer to potential customers.

Essentially, the following factors need to be taken into account when deciding on a location (Meyer-Stamer 1999; Kearney 2004):

- hard factors, such as proximity to buying and selling markets, transport connections, workforce, wage, and salary levels and/or local taxes;
- soft company-related factors, including business climate, contacts in the industry, partnership opportunities, universities, and research bodies, and
- soft employee-related factors, such as the housing situation, environmental quality, schools, social infrastructure, and recreational opportunities.

To evaluate sites, a variety of indices have been developed, many of them based on scoring models. One example is the Local Attractiveness Index developed by the global management consultancy A.T. Kearney.



Fig. 4.6 Part of Accenture’s global IT service center network (the world map is based on the image “BlankMap-World gray” by Vardion, licensed under Creative Commons CC-BY-SA 2.5)

To exploit the benefits of worldwide locations, providers such as Accenture and Cognizant, are setting up global IT delivery networks. One of the advantages is that the scale of onshore versus offshore activities can be chosen to suit each client. Another benefit is that a global network can be regarded in the light of a back-up strategy. The following map shows some of Accenture’s global IT delivery centers (Fig. 4.6).

Against this background, the development of *global delivery models* (Prehl 2006, p. 38) is becoming increasingly important. The term covers more than offshoring alone. Rather, it encompasses a mix of onshoring, nearshoring, and farshoring. Occasionally, this form of organization is referred to as next-generation offshoring—and given the creativity of marketing experts in coining new phrases, it is only a matter of time until we hear talk of Offshoring 2.0.

In particular, thanks to their global footprint, large, international software providers can break projects down and assign tasks to onshore, nearshore, or far-shore locations according to their specific requirements. Using Infosys as an example, Table 4.1 shows how a software development project can be distributed across various sites. Of course, the exact form of organization depends on the nature of the project in question.

An analysis of software companies’ operational and organizational structures reveals that an approach by which the technical specifications are drafted in a high-wage country, while code development and system testing take place offshore is increasingly a thing of the past. Even if this quasi-traditional method of dividing up tasks is still to be found among offshore projects, the trend is clearly heading away from a hierarchical approach to more egalitarian forms of co-operation. For example, at SAP, responsibilities are parceled out among the various sites by module. One result of this is that some development managers in Walldorf, SAP’s headquarters in Germany, now report to senior managers in India.

Table 4.1 Software development according to Infosys' global delivery model (based on Government of India 2006a, p. 131)

Onshore	Nearshore	Farshore
Architecture	Requirements analysis	Detailed design
Requirements	High-level design	Code development
Change management	Prototype building	Testing and integration
Implementation	Implementation support	

4.5 Outsourcing by Software User Organizations

In this section, we will examine current findings on outsourcing of development, modification and maintenance of application software from the software users' perspective. The results are based on a 2009 empirical study (Buxmann et al. 2010), which targeted IT decision makers from large, midsize, and small companies. Of the 5,500 companies contacted, a total of 498 usable questionnaires were returned. A wide range of industries were represented. The study's aim was to depict the current situation with regard to outsourcing of the task areas mentioned above. But the findings also reveal what user organizations require of the software companies and service providers they contract.

4.5.1 Outsourcing of the Development of New Custom Software

4.5.1.1 Take-up Rate for Outsourcing Services

The companies questioned in the study were first asked whether they are currently deploying a large-scale custom software system. This was the case with about 47 % of respondents. In comparison, around 53 % said they did not use a custom system, so that means they predominantly deploy standard application software (see Fig. 4.7).

Almost 80 % of the companies who deploy a custom software system have outsourced development work to an external service provider (see Fig. 4.8). The remaining organizations have not chosen to outsource. One-third of those who did so contracted out the entire development process. About two-thirds employed both external providers and in-house staff to develop the custom software (see Fig. 4.9).

We also looked separately at mission-critical versus nonmission-critical processes (Brandt 2010). The results are shown in Figs. 4.10 and 4.11.

4.5.1.2 Criteria for Selecting Providers to Develop New Custom Software

The research participants were then asked which criteria they use to select a partner to develop new custom software. This question was only addressed to

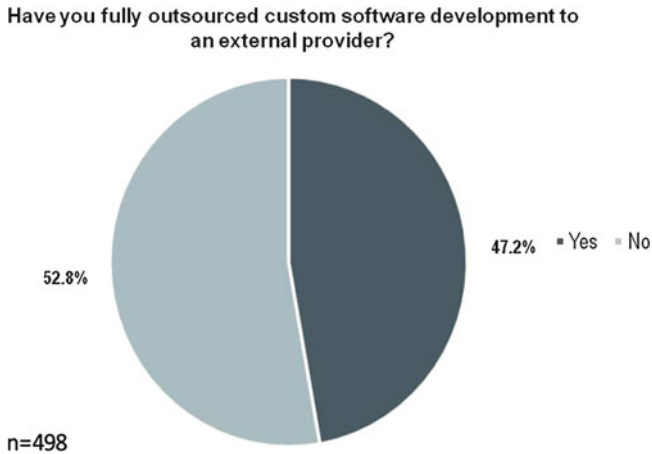


Fig. 4.7 Proportion of custom software systems

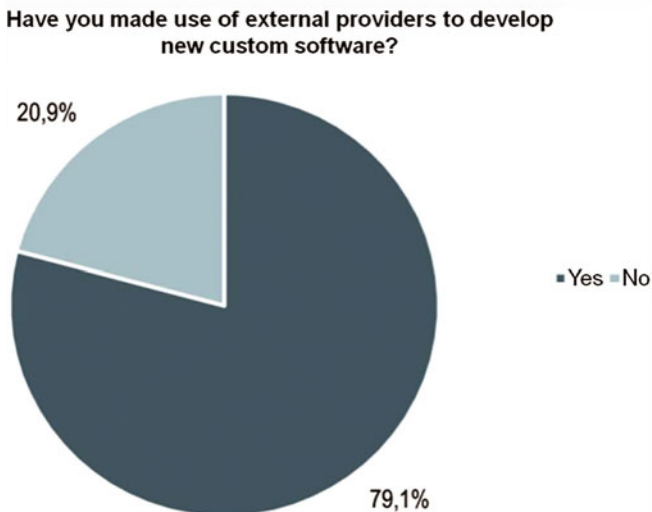


Fig. 4.8 Development of new custom software: share of companies who have outsourced

companies who have employed outsourcing services. The companies were asked to rate specific criteria (1 = very unimportant ... 7 = very important). Figure 4.12 shows how these criteria were ranked on the basis of the mean values. It is apparent that the companies we interviewed saw a long-term business relationship with the software company as the most important criterion.

One reason for this is that in the course of a long-term relationship, user organizations often task the software provider with the upgrading and maintenance of software solutions, too. References showing successful implementations in

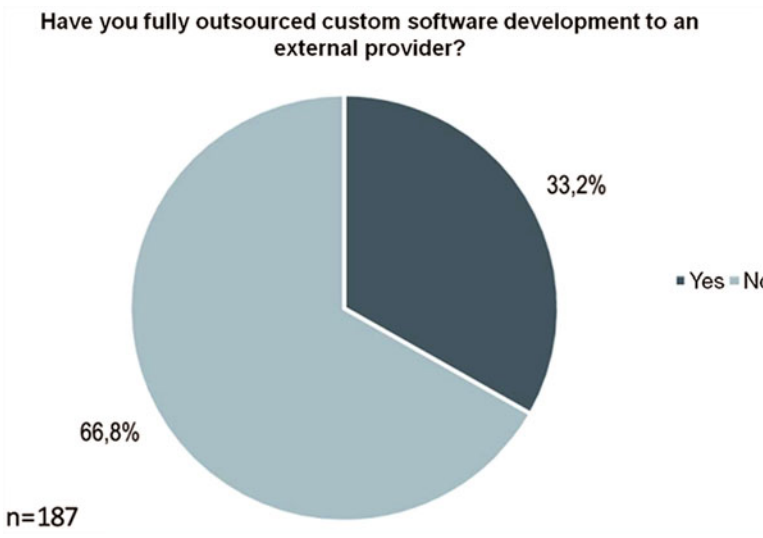


Fig. 4.9 Development of new custom software: share of companies who have fully outsourced custom software development

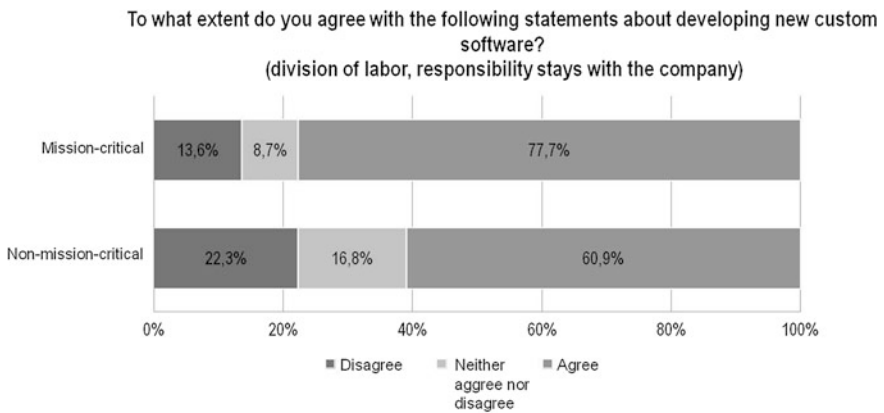


Fig. 4.10 Options for using outsourcing services (Responsibility stays with the company)

similar areas were cited as the second most important criterion. The quality of the provider’s employees presented to the company took third place. According to the respondents, the reputation of the software company also plays a significant role. The calculated mean of 5.11 shows that though value for money is not unimportant; it is not viewed as a top priority when selecting external providers. The companies expressed no opinion as to establishing a partnership of equals with the outsourcer. The purpose of this question was to discover whether SMEs tend to prefer providers of comparable size to themselves.

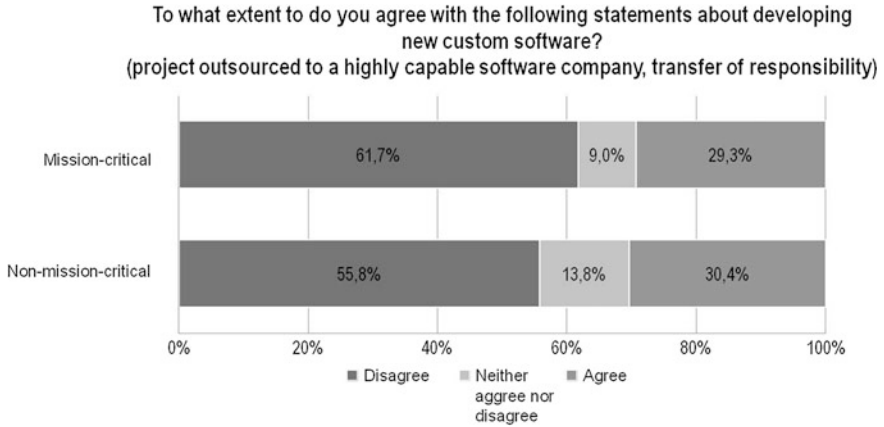


Fig. 4.11 Options for using outsourcing services (responsibility is transferred)

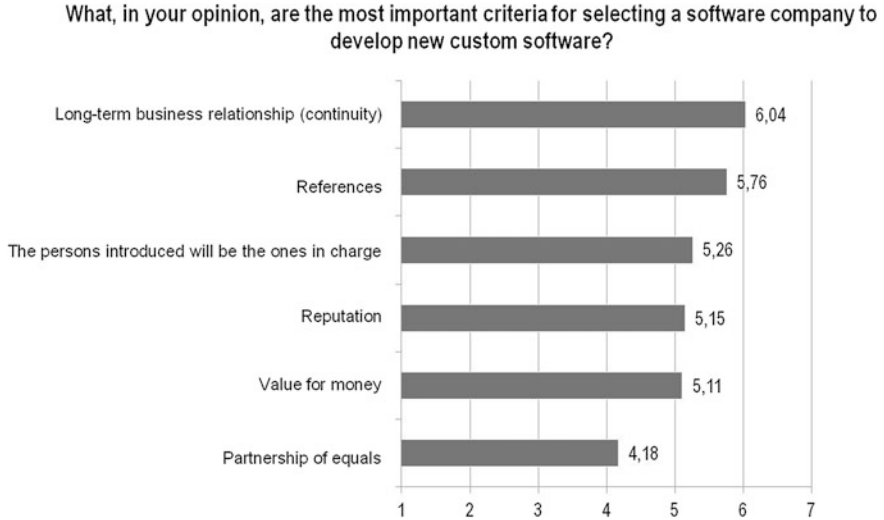


Fig. 4.12 Ranking of criteria for selecting external providers to develop new custom software (1 = very unimportant, ..., 7 = very important)

4.5.2 Outsourcing Modifications to Standard Software

4.5.2.1 Take-up Rate for Outsourcing Services

In this section, we will take a closer look at outsourcing of modifications to standard software. We have already identified the following alternatives (Buxmann et al. 2010):

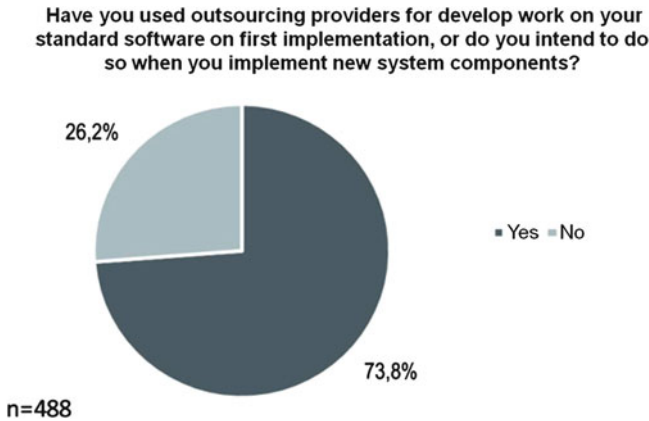


Fig. 4.13 Outsourcing of development work/modification of standard software

- Customizing/parameterization,
- Add-on programming, and
- Changes to source code.

Customizing and parameterization enable users to adapt the software to their business requirements to a relatively small extent. Extensive programming skills are not required. Customizing is part of the process of implementing practically any standard business software.

However, it is seldom possible to tailor standard software to a company's unique needs through customizing alone. This is particularly true for business processes that are mission-critical or create competitive differentiation. So-called add-ons which can be integrated with the standard product are developed to provide additional functionality. Standard solutions often include interfaces for this purpose. Add-on development does not entail modifications to the standard software. A third alternative is to modify the source code of the standard solution. This is only possible if the vendor makes the source code available to users. An advantage of customizing and add-on programming is that the company-specific parameterizations and additions remain in place when a new release is installed. If substantial changes are made to the standard system, however, this is frequently impossible.

Some three-fourths of participants have made use of third-party services for implementing standard software (see Fig. 4.13). About 70 % of these companies stated that they did not outsource the entire task (see Fig. 4.14). In contrast to this, just under 30 % of these companies handed over end-to-end responsibility for implementing a standard system or components of a standard system to a third party.

The results show that customizing and parameterizing are by far the most common methods of modifying standard software, followed by add-on development, while source-code modification is seldom performed.

Have you fully outsourced development work on your standard software on first implementation to an external provider?

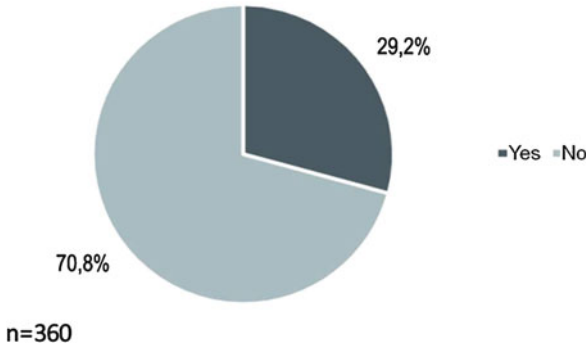


Fig. 4.14 Full outsourcing of development work/modification of standard software

What, in your opinion, are the most important criteria for selecting a provider of standard software to support your core business processes?

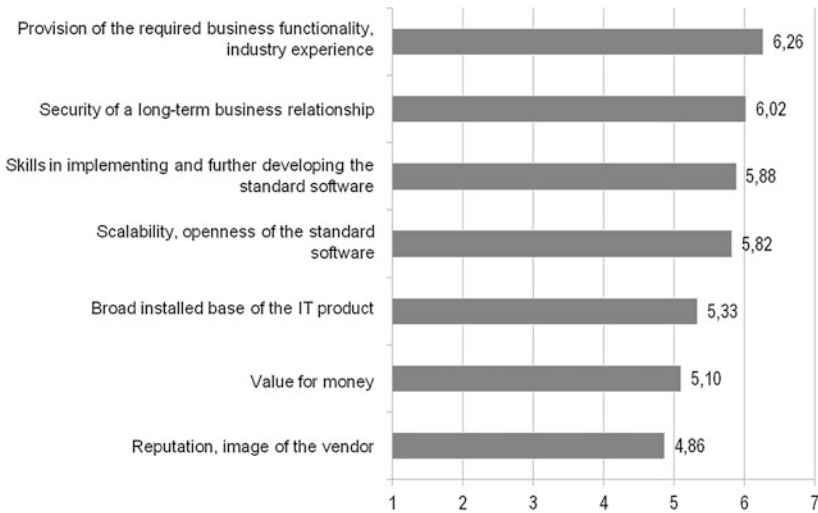


Fig. 4.15 Ranking of criteria for selecting a standard software provider

4.5.2.2 Criteria for Selecting Software Providers to Develop Standard Software

As with the evaluation of custom software providers, the participants were asked which criteria they applied in selecting standard software providers (Brandt 2010). Figure 4.15 shows how these criteria were ranked, again on the basis of the mean responses (1 = very unimportant ... 7 = very important). Provision of

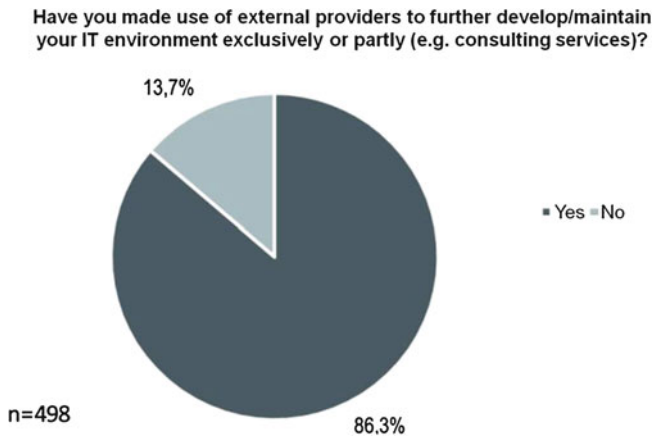


Fig. 4.16 Further development and maintenance: share of companies who have outsourced

the required business functionality and industry experience is the top priorities for users, followed by a long-term business relationship (continuity). The provider's skills in implementing standard software were also regarded as important by the participants who use standard software. The same applies to the software's scalability and openness. A broad installed base of the standard product is seen as relatively important. With regard to value for money, a similar picture emerges as with the selection of custom software providers: value for money is not seen as a top priority or a very important criterion in comparison to the others. The calculated mean of 5.1 shows that value for money is relatively important, but compared to the other criteria named, it is not seen as a high priority. With a mean of 4.86, the reputation of the software provider was given the lowest priority of all criteria.

4.5.3 Outsourcing the Further Development and Maintenance of Application Software

4.5.3.1 Take-up Rate for Outsourcing Services

User organizations were also asked whether they had made use of outsourcing for further development and maintenance of application software (Brandt 2010). More than 85 % of respondents answered in the affirmative (see Fig. 4.16). A minority handles these tasks entirely in-house.

The vast majority (72.6 %) of respondents, who have outsourced further development and maintenance of application systems, stated that they did not exclusively outsource but used third-party services to complement work done in-house (see Fig. 4.17). More than one-fourth of these companies *completely* outsourced these tasks to service providers.

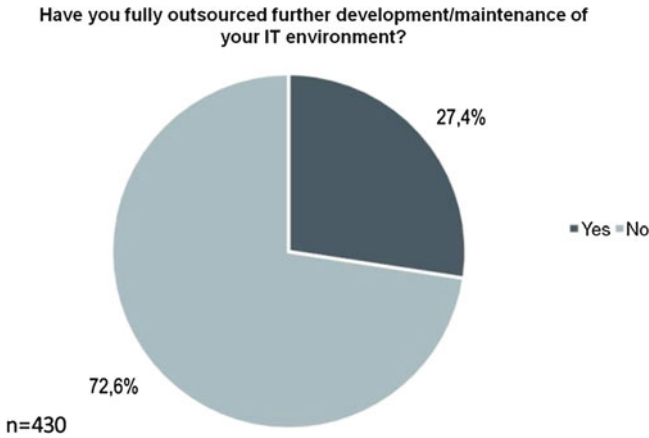


Fig. 4.17 Further development and maintenance: share of companies who have fully outsourced these tasks

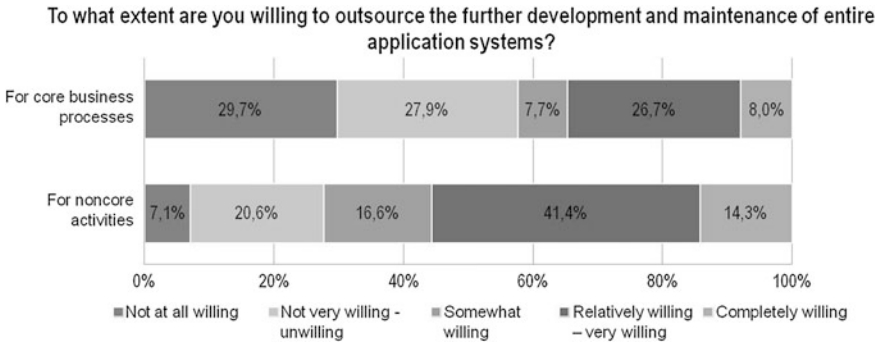


Fig. 4.18 Outsourcing of further development and maintenance of entire application systems

Participants were also asked whether, and to what extent, their willingness to outsource further development and maintenance of application software depends on whether the software solution in question supported mission-critical or less-critical business processes. The results show that companies are far more willing to outsource these tasks with regard to less-critical processes than core business processes (Fig. 4.18).

The results also indicate that back-sourcing, i.e., returning previously outsourced processes associated with further development and maintenance of application software is relatively uncommon to date. For example, more than 80 % of respondents asserted that they had not yet brought back in-house previously outsourced further development and maintenance of application software (see Fig. 4.19). Less than 3 % have done so. However, 15 % of companies were contemplating deploying their own staff (once again) to perform these tasks.

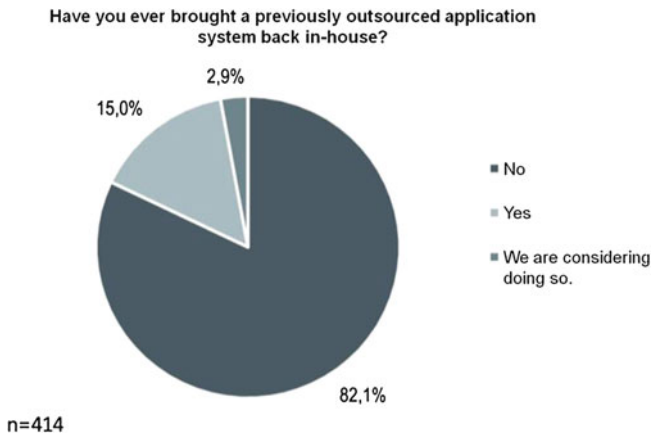


Fig. 4.19 Backsourcing the further development and maintenance of entire application systems

4.5.3.2 Selection Criteria for Software Providers

In the following survey, participants were asked what criteria they apply when selecting an outsourcing partner for the further development and maintenance of their application software. The question was only addressed to companies who have already outsourced to software and service providers. Figure 4.20 shows how the criteria were rated.

The results show that the key factor for survey participants was a long-term business relationship with the software and service provider (continuity). This is probably because a long-term collaboration enables the provider to get to know the company, which reduces communication effort. References about successful implementations of a similar work area were identified by the companies as the second most important factor. Respondents identified the reputation of the software and service provider, value for money, and the people who are introduced to them as relatively important. Furthermore, participants indicated that the service or software provider being a similar size to them is not important to relatively important. Collaboration with as few IT service providers as possible, who are not specialists but have broad skills in several fields, was also seen as not important to relatively important.

4.5.3.3 Location of Service Delivery

In light of the important roles that further development and maintenance play in the software lifecycle, the companies were also asked whether there are specific advantages associated with the service provider being located near at hand. Almost 72.8 % of the companies who had already outsourced their further development and maintenance agreed with the statement that it is extremely important for the service provider to be local (see Fig. 4.21). As many as 81.9 % of respondents stated that they prefer to outsource to an onshore provider. Furthermore, 75.9 % of

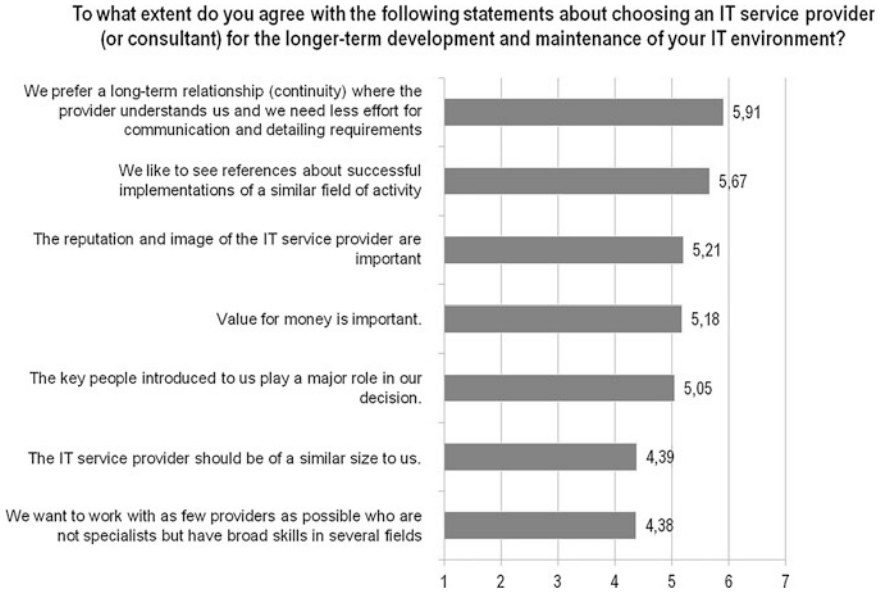


Fig. 4.20 Ranking of criteria for selecting a software/service provider to further develop and maintain an IT environment

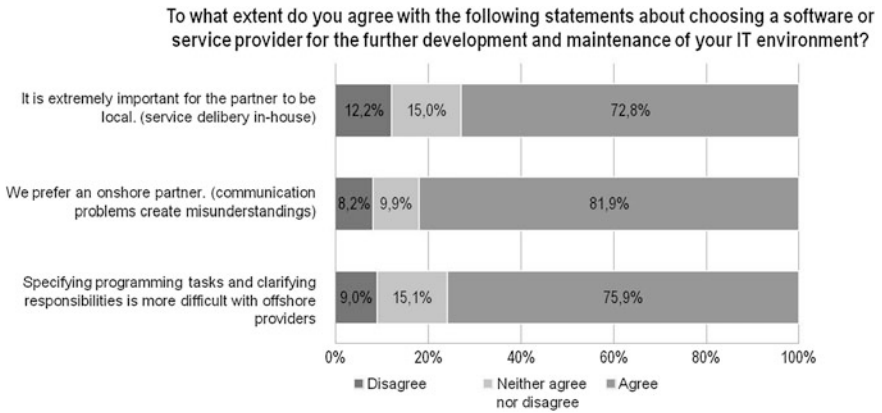


Fig. 4.21 Location of service delivery for the outsourcing of further development and maintenance of IT environments

the companies were of the opinion that it is more difficult to specify tasks and clarify responsibilities when collaborating with nearshore and farshore providers.

The above findings reveal a tendency toward outsourcing the further development and maintenance of application software to onshore providers.

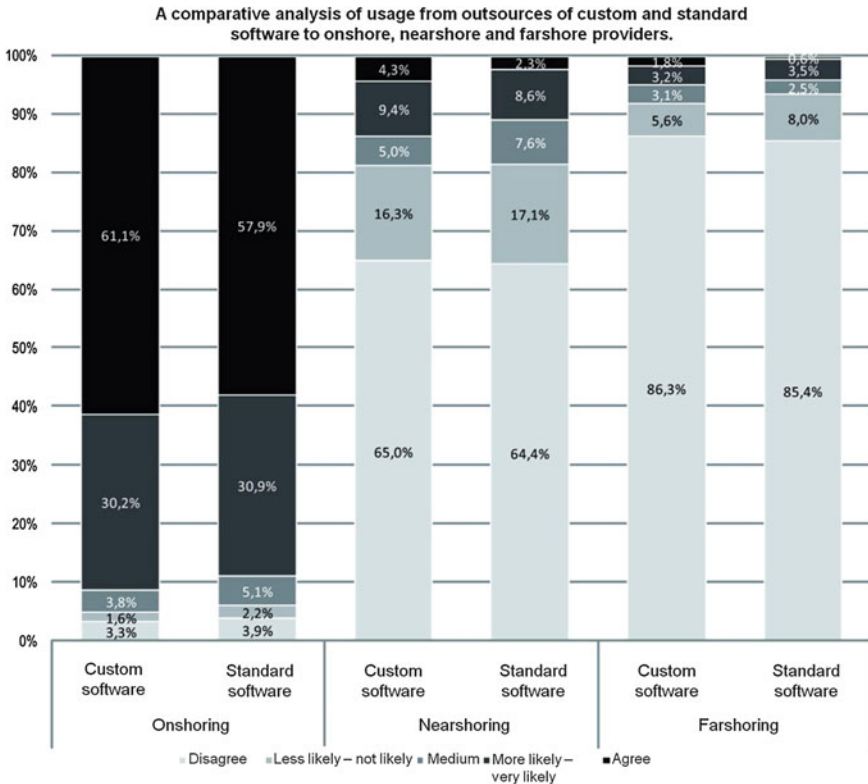


Fig. 4.22 A comparative analysis of usage from outsources to onshore, nearshore, and farshore providers for custom versus standard software

4.5.4 User Satisfaction with Onshore, Nearshore, and Farshore Providers

In the following section we will assess whether, and to what extent, the geographical distance between the provider and the customer affects the success of outsourcing projects (Buxmann et al. 2010).

Figure 4.22 shows the comparison of the response profiles regarding outsourcing the development of new custom software and the customizing of standard software to onshore, nearshore, and farshore locations. Interestingly, the responses were almost identical for all tasks.

How satisfied are user organizations with the services of their outsourcing providers? In order to find this out, we asked the companies about their experiences with onshore, nearshore, and farshore providers. We were particularly interested in how satisfied the companies were with the quality of products and services, costs, on-time delivery as well as communication and coordination. Figure 4.23 shows the mean responses. Only those companies from the sample

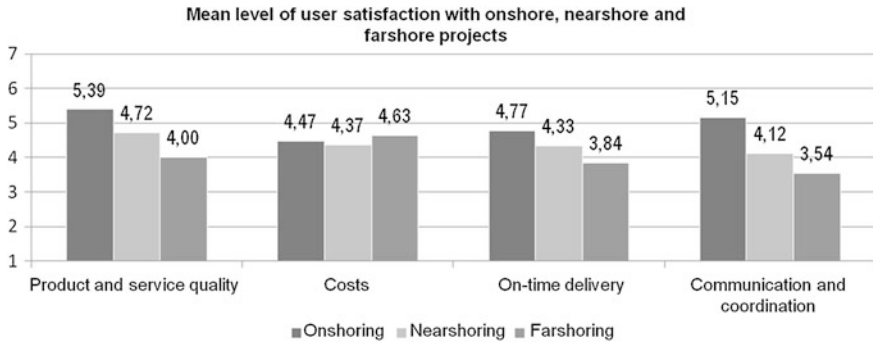


Fig. 4.23 A comparative analysis of user satisfaction with outsourcing services of onshore, nearshore, and farshore locations

who had already had experience with onshore, nearshore, and farshore providers were included. These tend to be the larger companies in the overall sample.

The results show that companies were most satisfied with onshore providers with respect to the quality of products and services, on-time delivery as well as communication, and coordination. Software and service providers in nearshore countries came out second best, whilst customers who used farshore providers were least satisfied on these counts. The results are statistically significant. In contrast to this, there were no significant differences in levels of satisfaction with costs.

A possible reason for these findings is that companies find the large geographical distance between themselves and farshore providers a problem. The main issue here is that geographical, linguistic, and cultural differences can make communication between the user organization and the provider much more difficult, which generally leads to inferior project results and/or projects taking longer to complete. These communication problems can be split into linguistic and cultural problems, which will be looked at in more detail in the following section.

4.6 Nearshoring Versus Farshoring: Distance from the Customer as a Success Factor?

4.6.1 Cultural and Language Barriers in Offshore Projects

Knowing how to work with people from different cultures is the success factor most frequently cited in studies of offshore projects (Buxmann et al. 2010; Gregory 2010). All of the experts we asked stressed the danger of cultural and linguistic problems leading to potential misunderstandings, a loss of trust, and lower employee motivation.

Due to India's status as an offshore location the cultural differences between Indians on the one hand, and Europeans and Americans on the other, are frequently discussed. In this context, different attitudes to hierarchies and the habit of skirting around bad news play an important role. Staff in India tends to seek the superior's

approval even for minor decisions and are sometimes reluctant to express their opinion, particularly in the presence of the project leader. They often find it difficult to say “no” even when it is necessary to do so. Furthermore, problems are not readily admitted to because that would lead to a loss of face. It is obvious that a question such as: “Can we rely on the prototype being delivered to schedule?” could cause difficulties in the project.

It is not only in India that such differences are evident. In their study of offshoring, Delmonte and McCarthy describe Russian developers’ reluctance to seek clarification of something they have not fully comprehended. This leaves customers with the impression that their requests have been understood, although this might not always be the case (Delmonte and McCarthy 2003, p 1609).

Cultural differences can even exist between neighboring countries. One expert recalled his experiences in the Netherlands. People are less formal with each other than in Germany and verbal agreements are generally no less binding than written ones. If a German employee asks for a verbal agreement to be put in writing, his Dutch counterparts regard this as showing a lack of trust and become suspicious.

Experts from the nearshore companies surveyed value the fact that the cultural differences between themselves and their German customers are small. If employees have more in common with each other, relationships tend to be more open in comparison to those with Indian or Chinese providers.

For communications to be effective, all parties need a certain level of language skills. Yet, this cannot always rule out mistakes being made due to differences of pronunciation, differences in the meanings of words, and in the technical terms used, which are all cultural by-products.

Offshoring and farshoring have been less widespread in Germany than in the UK or the USA, mainly due to the language barrier. German is still the working language of most companies although things are slowly but surely changing, particularly among major international players. It is understandable that the staff of German customers (unless they happen to be native speakers) prefers on the whole to use German. Thus wherever possible, exchanges with most of the German customers in development and first level support are conducted in German. Most of the experts we asked therefore attached great importance to language skills. As a result, global players are increasingly establishing onshore and nearshore locations and sourcing staff locally.

In a study of the relationship between outsourcing contracts and the success of the associated projects, Willcocks and Lacity discovered that projects tend to be more successful when the contract is detailed rather than loosely defined (Willcocks and Lacity 2006, p. 20 f.). Bearing in mind the language barriers inherent in offshoring, it is obviously imperative that contracts leave as little room for interpretation as possible.

Given that, almost all experts emphasize the significance of cultural and linguistic differences, it is worth addressing the question of how companies can meet this challenge. A common method is to provide intercultural training for both the onshore and offshore staff. Sometimes offshore providers even run such seminars for their customers.

By sharing their knowledge, experienced employees can make an important contribution to cultural awareness. That is why some providers send employees on exchange to other locations. One expert reported that it proved very worthwhile to invite two members of staff from India to work on projects in Germany. One of the many benefits this delivered was that the employees concerned were more committed to their work when they returned home. Other experts have observed similar phenomena. However, the preference is usually to send the customer Indian employees who have at least a basic knowledge of western corporate culture.

One expert said that he always advises his customers to send out an employee, likewise. By doing so, customers feel more in control and improve their intercultural skills at the same time. Many customers readily take up this advice.

Employing local staff is a third way to breakdown cultural barriers. One expert explained that even with projects in neighboring Austria or Switzerland, local employees were always included in the team, as they were better than their German counterparts at noticing nuances in communications with customers. Local staff often fills interfacing roles. Almost all offshore providers employ German staff for projects based in Germany. These individuals often head up the project and are the first point of contact for customers.

In the following section, we will consider the importance of face-to-face meetings in greater depth and examine to what extent these can improve provider/customer relations.

4.6.2 The Importance of Face-to-Face Meetings to Project Success

It is in the initial stages of a project that face-to-face meetings between the customer's and the software provider's employees have a major impact on the project's success. Most of the experts we surveyed agreed that it is better for a software development project to be partly carried out at the customer site. However, it is only the top international providers, with their nearshore development centers and multiple offshore locations, who are able to divide work up in a very granular way. Most of the experts interviewed assigned the different development stages as shown in Fig. 4.24.

The sequence shown in the above figure is only a general trend. For the success of a project, there needs to be some overlap between the project teams working on the various stages. The experts we surveyed stressed the importance of all participants working closely together throughout the project.

A large number of offshore providers maintain that about 30 % of time is spent onsite. Many experts emphasize the importance of keeping the onsite team, or at least some of its members, unchanged for the duration of the project.

The stages on the left hand side in Fig. 4.24 are rather communication intensive and should generally be carried out onsite with the customer rather than offshore. Even the latest communications technologies, such as video conferencing, are not substitute for meeting in person. It should be noted that, unlike providers, many German companies have yet to see video conferencing as an everyday tool. One

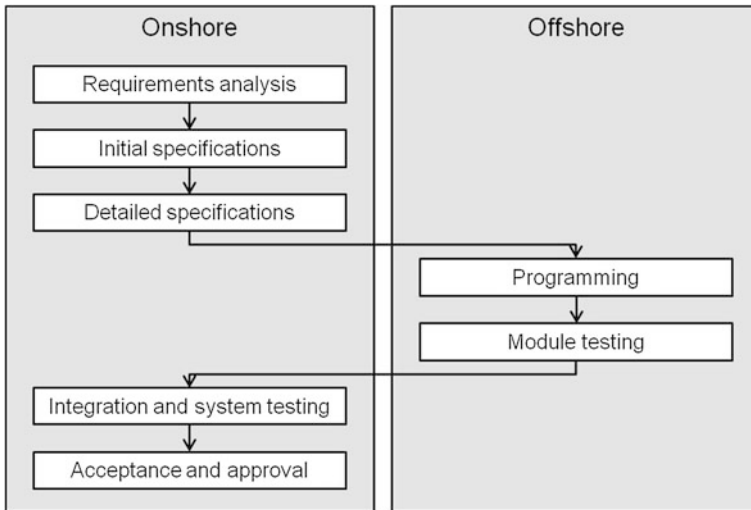


Fig. 4.24 Onshore and offshore division of the development stages

respondent related a situation in which customer and provider urgently needed to exchange notes as soon as possible. Because of organizational problems and the fact that it was not standard procedure, the video conference was not scheduled until a few days later, which wasted a great deal of time.

But if we assume that such means of communication will one day become a matter of course, then video conferencing could replace at least some face-to-face meetings and further boost the popularity of offshoring.

As the distance between provider and customer increases, so the travel times between them do. This will lead to customers more frequently opting for a nearshore location rather than a farshore location. Whether customers feel that solutions, such as documenting the project status online, are sufficient to overcome this drawback depends on the type of customer and project. Conversely, it tends to be the provider's employees, and not the customer's, who have to do most of the traveling.

It is ultimately the cultural distance, and not the mileage, between customer and provider that has a greater impact on the success of the project. All distributed project teams pose communication and coordination challenges, wherever they are located. And most respondents confirmed that the costs of communication and coordination in a development project do not rise in proportion with the physical distance.

4.6.3 Time Difference: Challenges and Opportunities

When there is a time difference between customers and offshore providers, the first potential difficulty is real-time communication such as telephone or video conferences. The greater the time difference, the smaller the window for these exchanges.

Offshore providers overcome this in two ways: by extending the working hours at offshore sites, and by planning this type of communication in advance. The experts we spoke to pointed out that in India, in particular, employees have few problems working long or unusual hours.

Aside from potential communication issues, time differences also create an opportunity—to apply the follow-the-sun principle described in [Sect. 4.3](#). In certain cases, this has worked successfully for American customers, for example. Requirements are gathered in the US during the day, and are then sent to India, where they are implemented as prototypes overnight. The next morning, results can be processed further at the US site, shortening both release cycles and development times, while increasing coordination costs, however.

From a European perspective, it is important to remember that time difference with India alternates between 3.5 h in summer and 4.5 h in winter, limiting the benefits of “following the sun”. An example of successful exploitation of time differences, however, is the acceptance testing carried out by Software AG in India (the establishment of the joint venture in Pune has already been discussed in [Sect. 4.2](#)):

Software AG’s software tests in India

Software AG maintains a testing team in Pune, India for the development of SOA-based solutions. The team takes the same approach as would customers intending to implement the solutions. It is responsible for the following tasks:

- Installing the product,
- Coexistence and integration testing,
- Verifying the interoperability of the product,
- Verifying basic functionality,
- Issuing error reports, and
- Retesting after corrections.

The team comprises 14 employees including the team leader. As well as the testing itself, it is their job to draw up test plans, to coordinate, and supervise testing activities, to deliver weekly progress updates, to train employees on the ground and to keep in contact with headquarters.

The development unit, which is distributed worldwide, provides complete product kits at regular intervals, which are then downloaded and tested by the team in India. The kits are produced overnight, so that the employees in Pune can begin testing first thing in the morning. When the working day begins at the development units the first results are already in. Queries can be answered and any reported errors resolved. According to Software AG, sharing work between multiple locations rather than working at the Darmstadt headquarters alone reduces costs by around 50 %. The costs of coordination have been factored into this estimate.



Fig. 4.25 Generic stages in software vendors' value chain (based on Wolf et al. 2010)

Another much cited advantage of time difference is the opportunity to offer round-the-clock support.

When we asked the experts what factors were key to facilitating work across time zones, they mentioned three in particular:

- Clearly defined processes,
- Integration of company sites, and
- Accurate documentation.

4.7 Outsourcing by Software Providers

Much has been written about how software users outsource IT tasks, and in [Sect. 4.5](#) we described a current study. A subject which has received far less extensive treatment, on the other hand, is the degree to which software and service providers themselves outsource tasks to specialized companies. We will focus on this issue in the following two sections: in [Sect. 4.7.1](#) we will review the current division of labor among software providers; in [Sect. 4.7.2](#), we will outline a new technology and discuss its potential impact on the structure of the software industry value chain.

4.7.1 The Status Quo of Specialization and the Division of Labor: Insights from Three Case Studies.

So far, no broad empirical studies on vertical integration in software companies have been conducted. Therefore, we will refer the study by Wolf et al. (Wolf et al. 2010; Wolf 2010), which investigated this issue at three mid-sized German vendors of complex standard enterprise software that requires extensive customization.

The starting point of this study is an eight-stage value chain in the software industry, as shown in [Fig. 4.25](#).

The study analyzed the situation from the software provider's perspective. Focusing on a particular point in time, it investigated whether activities within individual value chain stages were performed in-house, carried out by a partner company, or sourced from the market. In accordance with transaction cost theory (see [Sect. 2.4](#)), the first form of coordination is termed "hierarchy", the second "hybrid", and the third "market".

To visualize the scenarios we will use the “swim lane” concept (Binner 1987) supported by many process modeling tools. Based on the inter-company division of labor described in the case studies, each value chain stage is assigned to either the hierarchy, hybrid or market swim lane, or positioned between two of them. If a stage is positioned in the hierarchy swim lane, all corresponding subtasks are performed by the software provider, so there is no division of labor between companies. A stage positioned on the line between the hierarchy and hybrid lanes indicates a greater division of labor between companies, as one or more activities are hybrid in nature. From the software provider’s perspective, inter-company collaboration is most pronounced when a value chain stage is allocated to either the hybrid or market swim lanes or is on the line between these two; this signifies that the subject of the case study performs none of the associated activities. By comparing the state of affairs at two points in time, we can see whether the subject’s division of labor has changed over time, and where in the value chain this change occurred.

Case study 1 (Wolf et al. 2010)

This case study was conducted in May 2009 on an independent, mid-sized vendor of standard software. The value chain analyzed is that of an ERP solution for SMEs which offers functionality for merchandise management and production, financial accounting, business analysis, HR management, and customer relationship management and is designed to cover the business and accounting requirements of companies in the retail, manufacturing, and service sectors. The purpose of the study was to investigate the current situation and that at an earlier point of time, which the customer decided should be the year 2005.

In 2009, *product research* was primarily carried out by the subject’s own staff, although partners were involved for certain add-on products that broadened the product’s functional scope. In 2005, by comparison, all product research was conducted in-house.

Product development of the core product was carried out in-house at both points in time. However, as mentioned under product research, add-ons were included in 2009, and these were developed in cooperation with partners.

Both in 2005 and 2009, *documentation* was prepared in-house by specialists from the quality management department.

The company’s own staff took sole responsibility for generic *marketing* activities such as trade shows, advertisements, online media, branding, and brand campaigns—in 2009 and in 2005.

In both years, *sales* activities were carried out indirectly (see [Sect. 3.2.1](#)). End customers could not buy the product directly from the provider, but only from sales partners and software companies who in some cases specialized in specific product lines or industries. The subject had long-term relationships with these partners (in some cases up to 16 years). However, in 2009 some of the subject’s own sales staff were involved in pre-sales support and

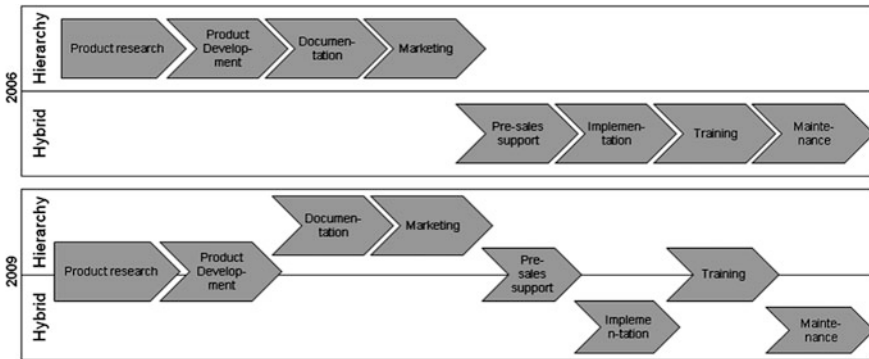


Fig. 4.26 Division of labor in case study 1 in 2005 versus 2009 (Wolf et al. 2010)

on request, assisted partners with customer presentations. This means that both hybrid and hierarchical coordination was employed in 2009.

Implementation was performed in both years by the partners mentioned in the context of pre-sales support; however in this instance, unlike the pre-sales support stage in 2009, the provider’s in-house employees did not provide support.

For the most part, *training*, particularly for new customers, was conducted by the partners responsible for pre-sales support and implementation. They trained the customer’s key users in the software, who, in turn, trained their colleagues. The subject only held training sessions under certain circumstances in 2009 (this did not happen in 2005). These sessions dealt with new releases or changes to the product regarding financial or payroll accounting, for example, as here the partner lacked the necessary skills to conduct the training.

Both in 2005 and 2009, the subject fully outsourced *maintenance* of the software product to its partners.

To summarize, Fig. 4.26 depicts the division of labor in 2005 and 2009, respectively.

Case study 2 (Wolf et al. 2010)

The case study was conducted in May 2009, at one of the leading mid-sized consulting and software companies for integrated ERP solutions in German-speaking Europe. The value chain analyzed is that of a modular, multilingual, multi-tenant, and multi-server ERP system for discrete, contract, and variant manufacturing. The study’s aim was to investigate both the current situation and the situation as it stood in the year 2000.

The subject of this case study did not differentiate between *product research* and *product development* but regarded them as a single stage.

Within the scope of product research, in-house staff was responsible in both 2000 and 2009 for devising new product ideas and analyzing customer requirements, and for passing these on to the development department. In terms of development, the company differentiated between application development per se and technical development (such as determining which tools should be used for software development). In 2000, technical development and development of the core product were carried out solely by internal employees. In 2009, the software was more open and offered various ways to access external programs such as Outlook, Excel, or Word from within the product. Also, basic functionality of the SQL database underlying the product could be used to display data in graphical form, for example, and this functionality appeared to the user to be an integral part of the product. However, as these were standard functions of Microsoft technologies deployed in the development process, we cannot treat them as program modules purchased from external transaction partners as defined by this study. As a result, this does not constitute hybrid coordination. In 2009, on the other hand, modules for HR management and customs freight documentation/clearing were integrated into the system, and these were developed by external software partners. We can conclude, therefore, that both hierarchical and hybrid coordination were present in the *product research* and *product development* stages in 2009.

To avoid the need to share specialist, architectural, programming, and documentation knowledge with too many people, *documentation* was compiled in both years by the internal employees involved in product research and development.

In 2009, the subject ran an in-house marketing department that handled press relations and CRM activities and communicated the brand image created by an external agency. In 2000, the subject was not yet collaborating with this agency, and hence the activities later performed by the agency were still done by in-house resources. Therefore, in the *marketing* stage of the value chain, both hierarchical and hybrid coordination could be observed in 2009, while in 2000 coordination was purely hierarchical.

In 2009, direct sales channels were deployed in the German market. In other countries, the subject implemented an indirect sales model with a small number of franchisees in which it had a minority shareholding. In 2000, on the other hand, the subject relied much more heavily on partners in the German, Austrian, and Swiss markets. However, this did not prove successful as the product was often not managed or sold properly.

Similarly, in 2009 the provider took full responsibility for implementation in the German market, and the franchisees mentioned previously performed these tasks in foreign markets. Partners were also responsible for implementation on foreign markets in 2000, signifying hybrid coordination. At this time, however, they also conducted implementation projects in

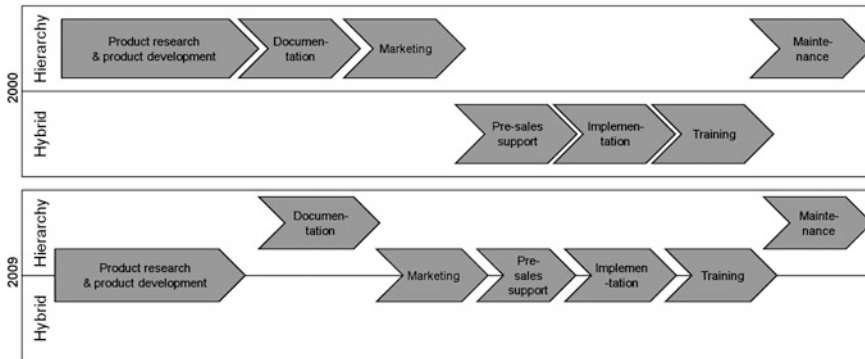


Fig. 4.27 Division of labor in case study 2 in 2000 versus 2009 (Wolf et al. 2010)

Germany. As the problems addressed by the software product tended to be highly customer specific, the subject did not operate a training department offering standardized seminars in either year.

On the whole, *training* was usually tailored to the needs of individual customers and carried out by the employees of the provider or of a partner company who performed the implementation. Therefore, in both years the forms of coordination correspond to those described for the implementation stage.

In 2009, the subject saw itself, particularly with regard to consulting-intensive projects, both as a supplier of its own software and as an integrator of other providers’ solutions. Therefore, in order to ensure customer satisfaction, it not only provided maintenance services for its own software but also for the add-on modules of various software partners, and for the third-party products managed by the company in its role as systems integrator. In 2000, the company did not yet provide maintenance for third-party modules, but already used in-house resources alone to maintain its own software product.

To summarize, Fig. 4.27 depicts the division of labor in 2005 and 2009, respectively.

Case study 3 (Wolf et al. 2010)

This study was conducted in April 2009. The value chain analyzed is that of an ERP software product for the plant and mechanical engineering sector and for car manufacturers and suppliers, comprising a standard ERP module and a range of modules addressing requirements unique to the above target industries. The study’s purpose was to examine the current situation and the situation as it stood in the period 1999–2001.

The product core was developed in a research project conducted around 25 years previously. At that time, the subject had been working with

customers to find a PPC system designed specifically for contract manufacturers. As there was no such solution available on the market, the subject collaborated with the customer and research organizations to develop a proprietary system. Excluding the product core (which already existed), the subject adopted the same approach to product research for its target industries (plant and mechanical engineering; car industry, in particular suppliers) in 2009. Add-on functionality was developed in collaboration with research organizations and integrated into the standard product. In light of the company's strategic positioning, the only requirement was that the functionality in question should be of direct relevance to manufacturing processes. With respect to make-or-buy decisions relating to the composition of the product portfolio, the subject basically pursued the following strategy: If the software related to a manufacturing issue that could strengthen the subject's position in this market, product research was largely conducted in-house. In these cases, the subject collaborated either with customers or with research organizations in the form of long-term partnerships, and the partners involved were selected on account of their industry specialization or geographic proximity. In contrast to this specialized approach, in 1999–2001 the company targeted a significantly wider market with its product. Because time to market was much shorter in this period, the subject's product research was conducted much more frequently in-house, and it focused more heavily on technical research projects. In summary, hierarchical and hybrid coordination could be observed in the *product research* stage of the value chain in both 2009 and 1999–2001.

In 2009, *product development* of the core functionality for the manufacturing process was primarily conducted by the company itself. However, its parent firm outsourced certain development activities to Poland. This Polish unit was a wholly owned subsidiary of the parent and provided services to its sister companies. However, seen from the subject's perspective, the involvement of the Polish operation was an example of hybrid coordination.

Furthermore, functionality in other, noncore areas not covered by the subject (such as document or quality management) were sourced from selected product partners specializing in certain kinds of solution. This add-on functionality was integrated via interfaces, and the subject acted as prime contractor for customer projects. Therefore, in 2009 both hierarchical and hybrid (Polish sister company) coordination were in place for core product development, while hybrid coordination (product partners) was the sole form when developing add-ons. In 1999–2001, on the other hand, the core product was developed exclusively in-house. Yet even at that stage, add-on functionality was already being sourced from third-party providers—to an even greater extent than in 2009. The company worked with far more third-party providers than in 2009, and these partnerships were not as close. To cover customer requirements, software was often sourced from the market and

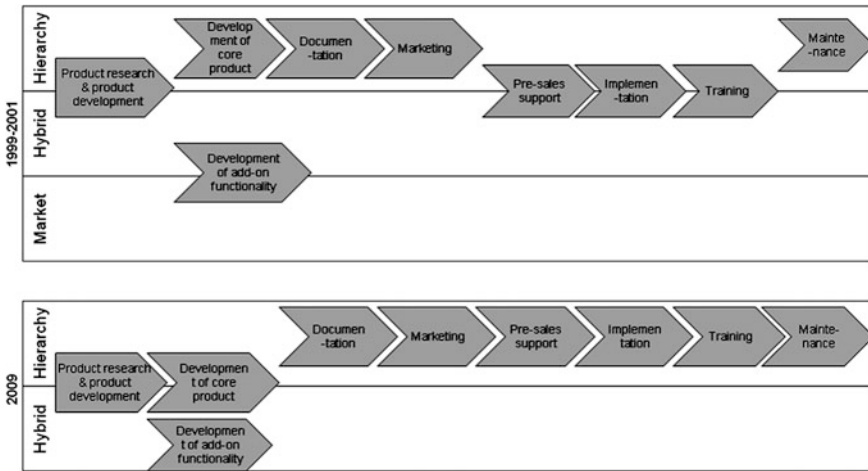


Fig. 4.28 Division of labor between companies for case study 3 in comparison (Wolf et al. 2010)

integrated on an ad hoc basis—corresponding to the market form of coordination.

For the period 1999–2001, therefore, we must make the following distinction: in terms of core-product development, coordination was exclusively hierarchical in nature; for add-on functionality, both hybrid and market coordination took place. In our visual representation, the “add-on development” subtask has moved from the line between the hybrid and market swim lanes in 1999–2001, to the hybrid swim lane in 2009. However, this does not represent a change in the division of labor from the subject’s perspective, as it was not involved in this subtask in either period.

In 2009, *documentation* was compiled in the form of wikis by employees from the marketing division, indicating hierarchical coordination. During this process, documentation for third-party products integrated in the core product was also incorporated into the wiki. Likewise, in 1999–2001 documentation was compiled solely in-house. However, wikis were not yet used, causing problems for the integration of third-party documentation, particularly in the case of online help.

Activities in the *marketing* stage of the value chain were performed almost completely by in-house resources in both periods.

In 2009, the main method of addressing the customer was via a direct sales channel, necessitated by the complexity of customer requirements and of the product itself. In 1999–2001, on the other hand, the subject targeted a wider market. As a result, the company also made heavy use of indirect sales channels via partners. In summary, hierarchy was the dominant form of coordination in pre-sales support in 2009; in 1999–2001, by contrast, both hierarchical and hybrid coordination were evident.

In 2009, *implementation* was more or less the preserve of the subject company. As for pre-sales support, partner companies played a larger role in 1999–2001 as the company was intentionally targeting a broader market segment, and as a result both hierarchical and hybrid coordination were evident.

In 2009, *training* was carried out at the customer site within the scope of implementation. In 1999–2001 on the other hand, the company ran an in-house training department at its main office, as it targeted a significantly broader segment in this period. Moreover, many of the company's partners also offered training. As a result, we can observe both hierarchical and hybrid coordination for this stage of the value chain.

The organization viewed *maintenance* as the most important source of revenue for standard software vendors, and as such these tasks were performed solely by the company itself during both periods.

To summarize, Fig. 4.28 depicts the division of labor in 1999–2001 and 2009, respectively.

Figure 4.29 provides an overview of the individual value chain stages, showing whether the software vendors in each case study experienced a greater (↑) or lesser (↓) degree of cross-company division of labor, or whether there was no change (×).

From this we can observe that inter-company division of labor increased, particularly for upstream stages of the value chain, product research in case studies 1 and 2, and product development in case studies 1, 2, and 3.

A comparatively low degree of inter-company collaboration was apparent for downstream stages of the value chain, for pre-sales support (case studies 1, 3), implementation (case study 3), and training (case studies 1, 3). In the first case study, the shift between 2005 and 2009 can be explained by two factors: additional support from partners in the pre-sales support stage for large and complex customer presentations, and the specialist training offered after certain new releases. In case study 3 on the other hand, the pre-sales support, implementation, and training stages were brought back in-house. This could be traced back to a strategic decision to narrow the target market, and the subsequent switch to a direct sales model.

4.7.2 Future Division of Labor in the Software Industry

4.7.2.1 Service-Oriented Architecture as the Technological Basis for Software Development

As described in the previous section, the division of labor between companies demands common business and technology standards. We will now present the service-oriented architecture (SOA) concept, which is based on open (technical) standards and can provide a foundation for inter-enterprise collaboration in the

	Product Research	Product Development	Documentation	Marketing	Pre-sales support	Implementation	Training	Maintenance
Case study 1: 2005 vs. 2009	↑	↑	✘	✘	↓	✘	↓	✘
Case study 2: 2000 vs. 2009	↑	↑	✘	↑	✘	✘	✘	✘
Case study 3: 98/01 vs. 2009	✘	↑	✘	✘	↓	↓	↓	✘

Fig. 4.29 Overview of case study findings

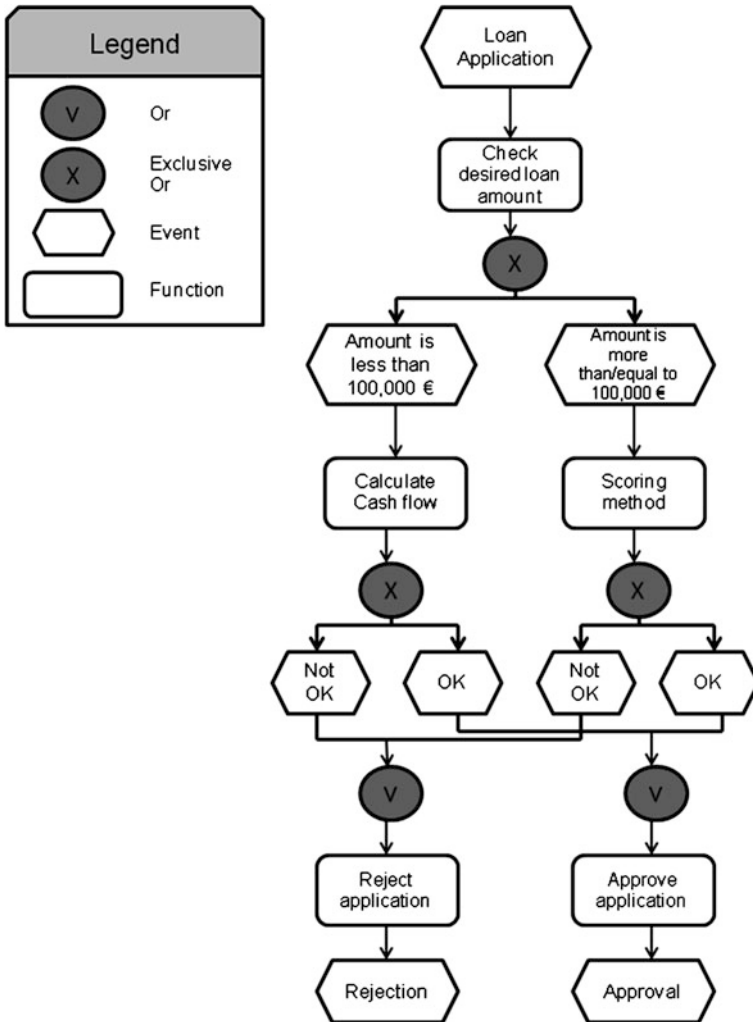


Fig. 4.30 EPC representation of the sample process

software industry. A system based on the SOA concept packages the functionality of business processes in the form of services. This means it is not merely a technical issue, but of increasing interest to management, too (Henn and Khan 2007). The principles of loose coupling and reusability are key to SOA: the former minimizes dependencies between services; the latter ensures certain functions need only be implemented once.

The idea behind this approach is to enable the creation and execution of business processes composed of modular services, either existing, proprietary ones, or from a third party. Each service performs a well-defined business task, and includes both the necessary data and corresponding business logic. Whereas we have previously discussed component-based programming as an abstract idea, these services are a concrete example of the technology in action.

We can illustrate this concept with a simple example: the business process of a financial services company, designed to evaluate the creditworthiness of business customers.

The following figure depicts the process as an event-driven process chain (EPC) (Fig. 4.30).

If a company applies for a loan of up to 100,000 euros, a simple credit assessment is used to calculate its cash flow. If the latter reaches a certain threshold, the application is approved; otherwise, it is rejected. If the loan requested is more than 100,000 euros, on the other hand, a complex scoring method is used, which includes a multitude of parameters such as additional balance-sheet KPIs, the customer's industry, holdings, and so on.

One fundamental decision the modeler or software developer must make relates to the granularity of the service. Calculating cash flow, for example, is a very fine-grained service; it is simply a small program with a few add-ons. However, the entire credit assessment process could also be mapped as a single, and therefore relatively coarse-grained service.

When deciding on the granularity, developers must weigh up a number of factors (Erl 2006). The more fine-grained services, the greater number of interfaces between them, increasing complexity, and negatively impacting performance. Very coarse-grained services, on the other hand, hinder reusability and result in less flexible IT systems.

Business processes can be comprised of existing elementary services or other business processes. The definition of the order in which the services are executed, and the parameterization of the services, is termed "orchestration". As is often the case, there are a number of competing standards in this sphere, including the XML-based Business Process Execution Language (BPEL4WS and BPEL), that is supported by certain vendors. Other providers, on the other hand, are trying to promote their own standards.

As previously mentioned, various definitions exist for the term SOA. Krafzig, Banke and Slama (2006) describe a service-oriented architecture as comprising the elements illustrated in Fig. 4.31.

As shown here, an application frontend initiates processes, controls their execution and receives the values returned by the calls. One example is a Web

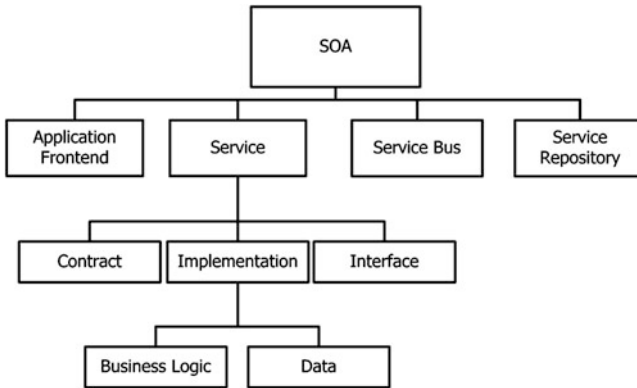


Fig. 4.31 Elements of an SOA (adapted from Krafzig et al. 2006, p. 57)

application in a company Intranet that employees use as a tool for deciding whether to approve loan applications.

Typically, a service represents a concrete business function, and according to Krafzig, Banke and Slama (2006) comprise three elements: contract, implementation, and interface. The contract provides information on the interface(s), the purpose of the service, what service quality can be expected, and what is required to use the service correctly. The interface presents the service to the user, while concealing its internal implementation. The implementation, at least in theory, contains both the business logic and the data necessary for modeling business functionality. In practice, however, business logic and data are still often separated, with the latter stored conventionally, in databases.

For the loan approval process described in this section, the “Calculate cash flow” function could be modeled as a service. The input data are processed by the business logic (in this case a formula for calculating the cash flow). The contract includes an interface description plus information on currencies and the accounting methods that the service supports. It can also be a repository of data on the reliability of the service, and who maintains it. Employees or other services can then use the service via a Web application (i.e., an application frontend).

The service bus supports communications between the application frontends as well as between different services, and must be able to handle the different characteristics of various actors. This it does by deploying multiple programming languages or operating systems, for example. It is sometimes necessary to translate messages reciprocally, into the syntax form (and semantics, if applicable) of the respective communication partner. In this regard, the role of the service bus is similar to that of hardware buses in PCs. Yet it can also offer additional services, such as logging, load balancing, and authentication of all actors, for example.

The service repository enables registration, search, and discovery of services. It also provides information on how they are used, including the service contract, version information, or the provider’s physical address, for example. The larger

the number of available services, the more important the repository becomes. Repositories do not have to cover only services available in-house; a truly visionary idea is to have globally accessible, cross-enterprise repositories which provide services that can be used and re-used in applications of many different kinds.

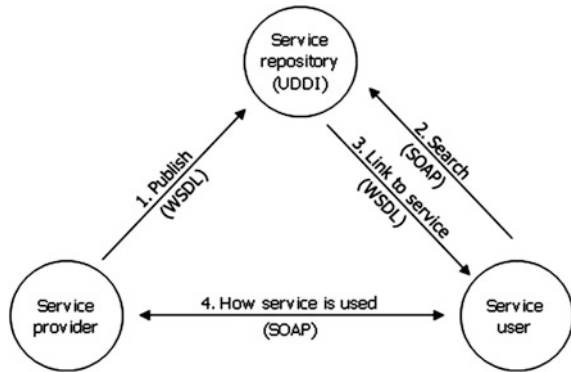
Let us now look at technologies and standards that can be used to implement the SOA concept. While SOA is frequently associated with Web services (Alonso et al. 2004), it is in fact an abstract concept that can be implemented using a variety of technologies. However, the most promising approach is an implementation based on Web services and the corresponding communication standards (UDDI, WSDL and SOAP). As these standards are supported by the SOA solutions of vendors like IBM, SAP, and Software AG, we will briefly examine this technology in the following section. Web services rely on an XML-based exchange of data. This open standard is particularly effective in enabling the IT systems of different companies to communicate with each other.

As with SOA, the term “Web service” is defined in a number of different ways. The World Wide Web Consortium (W3C) defines a Web service as a “software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.” (Booth et al. 2004, p.7). In the following section we will take a brief look at these standards.

SOAP is a XML-based message format used to exchange data between Web services. The *Web Service Description Language* (WSDL) is also based on XML. A WSDL document includes information on the Web service’s interface, and describes parameters and return values for the operations that the service supports. This section will describe the elements of WSDL in more detail, using a straightforward example. The *Universal Description, Discovery, and Integration* (UDDI) specification essentially describes how information on the services can be stored in the repository and how it can be retrieved. The following figure is a schematic representation of a SOA based on Web services; the standards used are given in brackets.

From registration to use of a Web service, the sequence can be described as followed: to register the service, the service provider sends the Web service description, in the form of a contract for example, to the UDDI-based service repository (step 1 in Fig. 4.32). The contract can include a WSDL document formally describing the interface. Moreover, it is possible to publish information on the service semantics in the repository. As soon as a potential user sends a message to the repository asking about the service (step 2), the corresponding WSDL document (step 3) is provided. With this information, the service user can now communicate directly with the service provider via SOAP messages (step 4). Service discovery and binding is possible both at the development stage and at runtime. However, the latter has mainly been a theoretical option up to now.

Fig. 4.32 Schematic representation of a SOA based on Web services (adapted from Dostal et al. 2005, p. 28)



Let us now illustrate what we have just described using an example, how the “Calculate the cash flow” function can be performed using a service. The service is based on the following Java class, which implements a simple cash flow calculation method (Fig. 4.33).

This Java class can be offered as a Web service by means of a framework such as Apache Axis, making it possible to automatically generate a corresponding WSDL document. Before taking a closer look at this document, we will first describe the main language artifacts of WSDL (Alonso 2004, p. 165-174).

A WSDL document describes the Web service both in abstract terms (<port types>, <operations> and <messages>) and in concrete terms (<binding>, <port> and <service>). The <port type> command combines related <operations> into sets. Each <operation> is specified via its incoming and outgoing <messages>. The commands mentioned so far only provide an abstract description of the Web service; they do not include details on the concrete physical address or protocols used. The <binding> command describes the communication protocols (e.g., SOAP) that use a <port type>. The <ports> can state the concrete address of the Web service in the form of a uniform resource identifier (URI). The <service> command, finally, defines a set of <ports>, which means a given service can be offered to a range of physical addresses. The following figure shows the WSDL document that was generated from the Java class to calculate cash flow. The elements described above are shown in bold (Fig. 4.34).

We would like to emphasize again at this point that SOA is not tied to the Web service technology. Other technical implementation options are also conceivable, on the basis of DCOM or CORBA, for example.

4.7.2.2 Potential Implications of SOA for the Value Chain Structure

After outlining the basic concept of SOA, let us now discuss its potential implications for the structure of value chains in the software industry. The Web service for calculating cash flow, for example, could be used both in credit rating software and in a program for evaluating company value. The automotive industry already takes a similar approach, using individual modules or systems in as many different

```

public class CashFlowBerechnung {
    public double cashFlowDirekt (double fWERträge, double fWAufw,
        double fWKSbErträge, double fWKSbAufwend) {
        double CfD = fWERträge - fWAufw + fWKSbErträge - fWKSbAufwend;
        return CfD;
    }
}
    
```

Fig. 4.33 Java class to calculate cash flow

<pre> <?xml version="1.0" encoding="UTF-8" ?> <wsdl:definitions targetNamespace="http://localhost:8080/axis/ CashFlowBerechnung.jws" xmlns:apachesoap="http://xml.apache.org/xml- soap" xmlns:impl="http://localhost:8080/axis/CashFlowBerechnung.jws" xmlns:intf="http://localhost:8080/axis/CashFlowBerechnung.jws" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:wsdlsoap="http:// schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/ XMLSchema"> </pre>	Definition of namespaces
<pre> <wsdl:message name="cashFlowDirektRequest"> <wsdl:part name="fWERträge" type="xsd:double" /> <wsdl:part name="fWAufw" type="xsd:double" /> <wsdl:part name="fWKSbErträge" type="xsd:double" /> <wsdl:part name="fWKSbAufwend" type="xsd:double" /> </wsdl:message> <wsdl:message name="cashFlowDirektResponse"> <wsdl:part name="cashFlowDirektReturn" type="xsd:double" /> </wsdl:message> </pre>	Description of message formats <message>
<pre> <wsdl:portType name="CashFlowBerechnung"> <wsdl:operation name="cashFlowDirekt" parameterOrder="fWERträge fWAufw fWKSbErträge fWKSbAufwend"> <wsdl:input message="impl:cashFlowDirektRequest" name="cashFlowDirektRequest" /> <wsdl:output message="impl:cashFlowDirektResponse" name="cashFlowDirektResponse" /> </wsdl:operation> </wsdl:portType> </pre>	Description of operations <portType> <operation>
<pre> <wsdl:binding name="CashFlowBerechnungSoapBinding" type="impl:CashFlowBerechnung"> <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/ soap/http" /> <wsdl:operation name="cashFlowDirekt"> <wsdlsoap:operation soapAction="" /> <wsdl:input name="cashFlowDirektRequest"> <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/ encoding/" namespace="http://DefaultNamespace" use="encoded" /> </wsdl:input> <wsdl:output name="cashFlowDirektResponse"> <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/ encoding/" namespace="http://localhost:8080/axis/ CashFlowBerechnung.jws" use="encoded" /> </wsdl:output> </wsdl:operation> </wsdl:binding> </pre>	Description of message transports <binding>
<pre> <wsdl:service name="CashFlowBerechnungService"> <wsdl:port binding="impl:CashFlowBerechnungSoapBinding" name="CashFlowBerechnung"> <wsdlsoap:address location="http://localhost:8080/axis/ CashFlowBerechnung.jws" /> </wsdl:port> </wsdl:service> </wsdl:definitions> </pre>	Web service address <service> <port>

Fig. 4.34 WSDL description of the Web service (style: RPC/encoded)

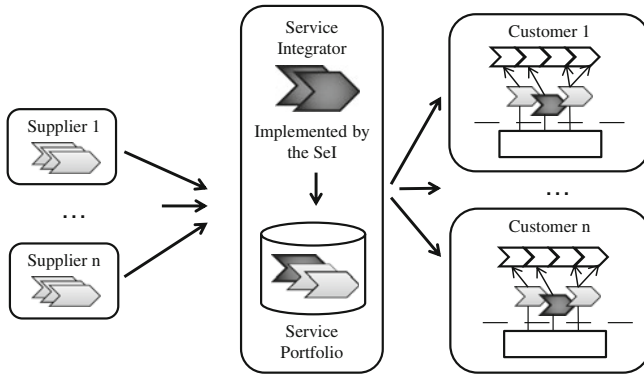


Fig. 4.35 Example of a supply network in the software industry of tomorrow

products as possible. Service repositories, e.g., based on UDDI, are a good way of facilitating the use and reuse of services.

Thanks to common open standards such as SOAP, WSDL, or UDDI, the service-oriented paradigm is opening up new opportunities for collaborative software development and for greater specialization (reducing the scope of activities performed in-house). For example, a standard software provider could outsource development of certain services to offshore locations and integrate services from smaller providers into its products to provide niche functionality. To return to value creation in the automotive industry: smaller software companies could take on the role of suppliers, while the ERP system provider assumes the role of a service integrator, providing the customer with made-to-measure, service-based applications that support their unique business processes (see Fig. 4.35).

Services are integrated via an SOA platform. These are developed by big-name players such as IBM, Microsoft, Oracle, or SAP, but also by smaller providers such as Software AG.

In Chap. 5, we will take a closer look at the structure and economic characteristics of platforms.

5.1 Overview

The rise of platform concepts can be observed in numerous industries. The automotive industry was already hailing product platforms as a recipe for success back in the 1990s. Many other industries followed suit (cf. e.g., Köhler 2004 on product platforms in the media industry). In the following, we will explore the role of platforms in the software industry. Drawing on the work of Gawer (2009), we can distinguish between two generic types of platforms: product platforms enable products and services to be produced efficiently through the re-use of existing modules (Wheelwright and Clark 1992); while the main purpose of industry platforms is to attract complementary products and/or services from third parties in an industry (Cusumano and Gawer 2002). Figure 5.1 shows other characteristics of these two types.

Both varieties of platforms are now also found in the software industry. In the field of consumer software, there is already a multitude of examples: the best known is Apple's AppStore for the iPhone. Industry platforms for business software are still in their infancy, for example salesforce.com's AppExchange, Google Apps Marketplace, Microsoft Pinpoint, or SugarCRM's SugarExchange (Burkard et al. 2010).

5.2 Product Platforms in the Software Industry

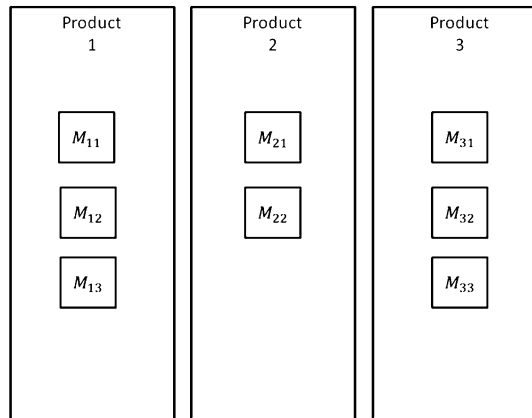
5.2.1 Cost Structure of Platform-Based Software Development

Like their high-profile counterparts in the automotive industry, product platforms in the software industry are intended to reduce development costs while maintaining or enhancing quality. In particular, software product platforms are used to make and deliver products in a product line that address different target groups with different price categories and system environments—flexibly and above all efficiently (Reussner and Hasselbring 2006).

Fig. 5.1 Platform typology (based on Gawer 2009)

Platform Typ	Produktplattform	Branchenplattform
Participants	One company (and in some cases its suppliers)	Multiple companies with compatible products
Objectives	<ul style="list-style-type: none"> • Faster and more efficient development and production • Greater product variety at lower cost • Increased flexibility when designing new products 	<ul style="list-style-type: none"> • For platform operators: Boost a platform's usefulness through complementary products and services • For suppliers of complementary services: Increased sales
Design principles	<ul style="list-style-type: none"> • Reuse of components • Stable underlying architecture 	<ul style="list-style-type: none"> • Stable interfaces for extensions

Fig. 5.2 Example of the development of three products without a platform



These product platforms are designed to exploit reusable software modules, just as the automotive industry example cited above involves the reuse of components (Boysen and Scholl 2009). A software module is a unit of software that performs a defined task or function and can communicate with other modules via interfaces, for example on the basis of the SOA standards outlined in the previous chapter, but can also be run independently. Modules are managed via configuration software, in other words the product platform. Most crucially, a module should be used in as many products as possible (Baldwin and Clark 1997; Miller and Elgård 1998; Meyer and Lehnerd 1997).

A simple cost model demonstrates how the introduction of a product platform can alter a software provider's cost structure. We shall assume that each of the provider's products is developed separately, but in a modular way. Accordingly, all N products, each consisting of M_n ($n = 1 \dots N$) modules, are developed independently of each other. Figure 5.2 illustrates an example with three products, each comprising two to three product-specific modules.

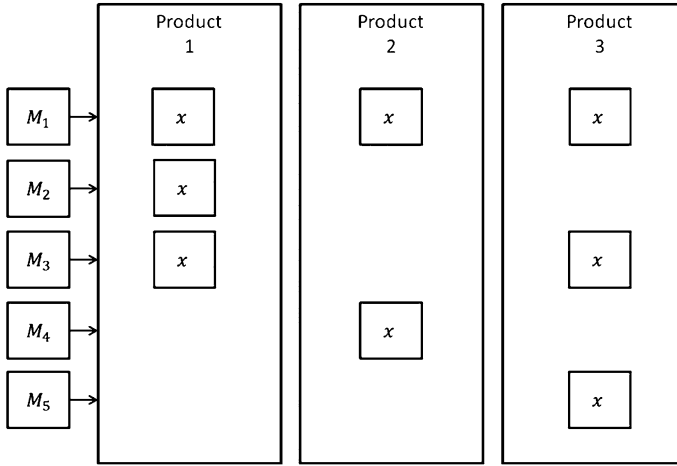


Fig. 5.3 Configuration of three products via a product platform

The system n has development costs K_n , which are made up of the total of the K_{nm} development costs of M_n individual modules ($M = 1 \dots M_n$). The costs of the development of the provider's N systems, therefore, amounts to:

$$K = \sum_{n=1}^N K_n = \sum_{n=1}^N \sum_{m=1}^{M_n} K_{nm}$$

This approach will now be contrasted with the cost structure after the introduction of a product platform, or in other words, the reuse of components. For the vendor, there will be upfront expenditure to establish the product platform. We will call this K_p . Development of the modules incurs costs for each of the D modules, which we will transcribe as K_d ($d = 1 \dots D$). Each module will be implemented in $1 \leq n \leq N$ systems. Integrating all these modules into a single system creates additional costs. To keep this simple, we shall assume an average cost rate, which we shall call K_d . Accordingly, the development of a vendor's N systems using a platform concept entails the following costs:

$$K = K_p + \sum_{d=1}^D K_d + N \times K_1$$

Figure 5.3 depicts a usage matrix showing which modules are used in which products. In our example, module 3 is used in products 1 and 3; module 1 is included in all three products. Modules 2, 4, and 5, on the other hand, are used in only one product.

By comparing cost functions (1) and (2), we can draw conclusions about the changes in cost structure that can be expected on the introduction of a product platform. First, we can clearly see that introducing a product platform requires a

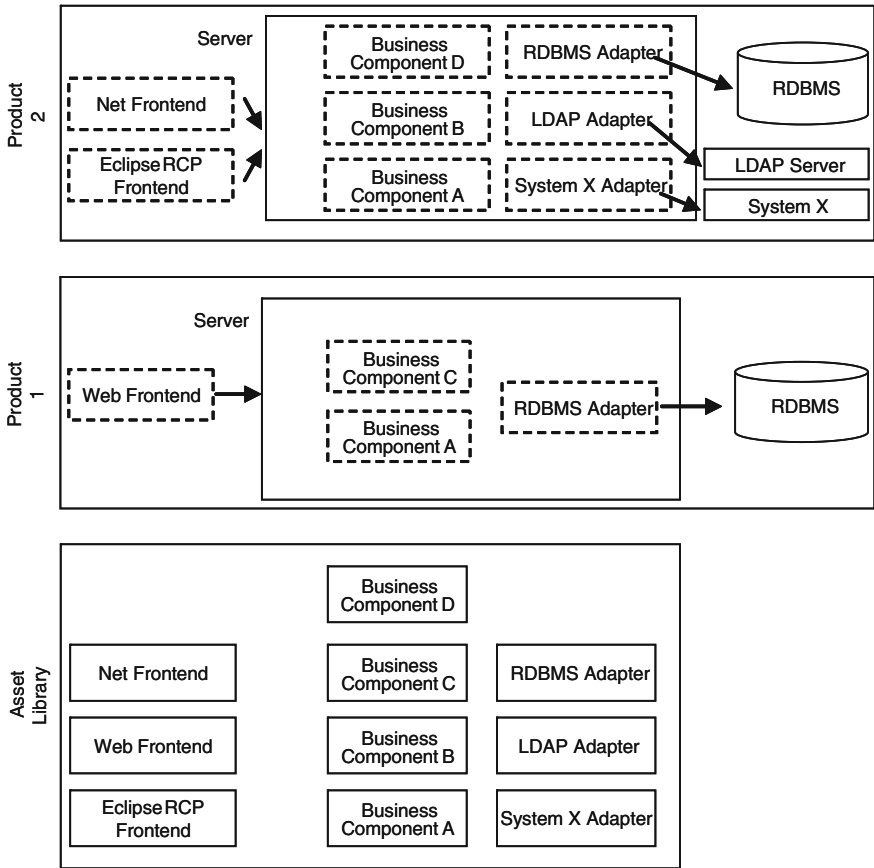


Fig. 5.4 Relationship between product and components (based on Zacharias 2007, p. 74)

large and potentially risky investment to develop it. For each of the N products there are integration costs of K_i . In real life, the size of these costs is determined by the quality and the future-proof design of the platform interfaces. Initially, at least, this upfront investment is a certain challenge.

However, the deployment of product platforms leads to sizeable cost savings, particularly if the module reuse rate is high. In other words, if the modules, once developed, can be reused in as many products as possible, or if the number of modules required to create for a given number of products can be reduced. Our example illustrates the latter scenario: the number of products remains the same, but the number of modules has been trimmed from eight to five.

In the automotive industry, as cited above, the leading manufacturers produce only a small proportion of the modules themselves. Instead, they concentrate mainly on integrating modules to create their products. The development and production of modules is the task of highly specialized suppliers.

5.2.2 Organizational Support for Implementing Product Platforms

A major challenge for software providers is to align their organizational structures with the rather abstract notion of a product platform. Here, both theory and practice are still embryonic. In addition to the simpler concepts found in the Web application development space, there are also more comprehensive approaches, such as component-based software development (Messerschmitt and Szyperski 2003, p. 244–263).

Zacharias provides some interesting initial insights (Zacharias 2007), dividing tasks into domain engineering, application engineering, and product line management:

- *Domain engineering* constitutes the technical, specialist, and organizational basis for component-based development. In particular, this includes the maintenance of an asset library, which contains all available components. Domain engineering also involves the provision of development environments and procedural models.
- *Application engineering* develops software solutions, drawing on the components provided by domain engineering, and with new, complementary components. Simultaneously, application engineering passes these newly developed components back to domain engineering.
- *Product line management* oversees domain engineering and application engineering and ensures these are in alignment.

In the final analysis, products are nothing more than special configurations of components, which are linked at the time of development, installation, or at runtime. Figure 5.4 illustrates the concept of component-based engineering, after Zacharias.

Van der Linden et al. (2004) have developed a guideline for the implementation and evaluation of product platforms. They look at a product platform from four perspectives: the business dimension, the architectural dimension, the procedural dimension, and the organizational dimension. For each of these four dimensions, there are different evaluation levels that can be used to assess an organization's current status in relation to the implementation of a product platform.

5.2.3 Add-on: Industrialization as a Management Concept for the Software Industry

In the previous section, we looked at component reusability and the related principle of standardization. In Chap. 4, we discussed various forms of outsourcing. In Sect. 3.4.2, we analyzed current approaches to automating software development. These three concepts are all part of the broader concept of industrialization, which is currently the subject of much discourse and is becoming increasingly important for the software industry.

Industrialization is a historically evolved management concept that offers a framework for cost-effective mass manufacturing. According to this concept, the key factors are: increased standardization of products and processes; greater specialization (i.e., an increasing division of labor); and automation (Heinen 1991, p. 10; Schweitzer 1994, p. 19). These three elements are interrelated, standardization being the most important prerequisite for implementing both specialization and automation. To put it another way: specialization and automation are not possible without standardization. But even standardization alone can lead to lower unit costs. This is because only standardized processes can be carried out by a machine or divided among multiple parties. Similarly, standardized processes are only possible with standardized products.

New technologies play a central role in industrialization. As they create new potential for automation, specialization, and standardization, they are often described as drivers of industrialization.

Based on these technologies, we can demarcate three stages of industrialization, two of which have already come to a close, and one that continues to this day (Condrau 2005). The first period begins with the Industrial Revolution, from 1780 onwards. The invention of the steam engine, railroad, and power loom saw artisan-type one-at-a-time manufacturing replaced by an early form of industrial mass production. Whereas goods had previously been made individually for the craftsman's own use, machines enabled the more standardized manufacture of large quantities of products for sale. This first stage of industrialization was a time of enormous productivity gains and rapid economic growth.

The concept evolved further during the second stage of industrialization, which commenced in 1840. In addition to studies conducted by academics, Taylor's work on scientific management is a case in point, key drivers in this phase were the discovery of electricity and the invention of electric motors. While in the first stage of industrialization, the production process was shaped by a quantitative division of labor (i.e., dividing similar activities between multiple machines or human resources), the installation of conveyor belts and the consequent increase in standardization meant that work could be divided according to the type of task. As a result of this greater specialization, highly standardized material goods, such as the Model T Ford, could be created, stimulating further productivity gains.

The IT revolution, which began in the late twentieth century, is now seen as ushering in a third stage of industrialization. While the first two stages involved the manufacture of material goods, the focus is now on creating information-intensive services and products. This development has been driven by the well-known waves of innovation in information and communication technology. Figure 5.5 summarizes the factors behind the industrialization of software development which were discussed previously in various parts of this book.

Figure 5.6 outlines the three stages of industrialization and their key characteristics. In the literature, the service sector is sometimes presented as the primary target of the third stage of industrialization. We do not share this point of view: Industrialization, through modern ICT, is also impacting information products such as books or software, which are not part of the service sector.

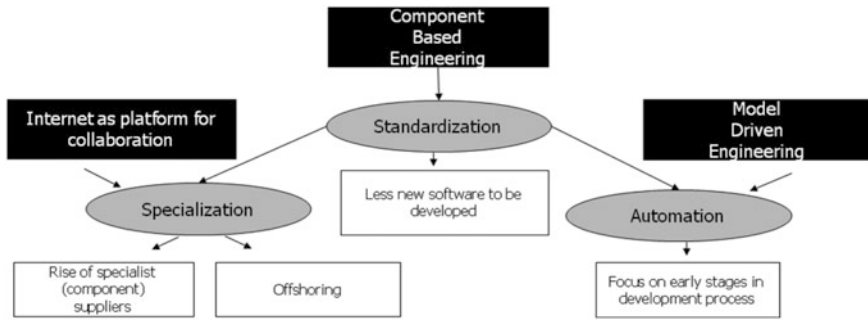


Fig. 5.5 Drivers and expected effects of industrialization in the software industry

	1st stage of industrialization	2nd stage of industrialization	3rd stage of industrialization
Drivers	Steam engine, railroad, power loom	Electricity, electric motors	Information and communication technology
Sector	Material goods	Material goods	Information-intensive products and services
Time period	Approx. 1780-1840	Approx. 1840-1960	Since the late 20 th century
Keywords	Industrial Revolution	Fordism and Taylorism	IT Revolution

Fig. 5.6 Three stages of industrialization

However, it should be noted that there are limitations to the division of labor and automation, and therefore to the concept of industrialization. These limitations concern the motivation of employees, the costs of distributed manufacturing, and the flexibility of the production processes. When there is a very deep division of labor, tasks become increasingly similar. Taken to excess, this becomes mind numbing and monotonous for workers. As a result, their motivation sinks, diminishing the unit cost advantage. In addition, a high level of automation requires a high level of standardization: This makes changes to processes, and therefore also to products, increasingly difficult. In light of both of these aspects, the manufacturing industry no longer sees maximum industrialization as its ultimate ambition.

5.3 Industry Platforms in the Software Industry

5.3.1 Openness of Industry Platforms

As we noted at the start of this chapter, industry platforms provide a basis for products and/or services whose functionality is extended by complementors. For example, most game consoles include only the basic functionality required for playing games, such as a graphics accelerator. The games themselves are developed by third parties. Industry platform operators' main objective is to maximize the attractiveness of their platform, by offering the largest possible number of complementary products to the largest possible group of end customers (who can be consumers or companies).

In more general terms, platform operators are primarily concerned with achieving the right degree of openness for their platform. A platform can be described as completely open, if it has no limitations as to participants, development, use, and commercialization of the platform (Eisenmann et al. 2009). This extreme form is rare, however, especially when it comes to commercially operated platforms. Rather than simply taking an either-or decision, platform operators need to find the ideal degree of openness for their platform.

The difficulty herein lies in the trade off between openness and control: while more openness encourages third parties to participate, increasing the platform's attractiveness, it will generally lead to a loss of control over the platform design (West 2003). In the following section, we distinguish between vertical and horizontal openness.

5.3.1.1 Vertical Openness

Vertical openness refers to what extent complementary products or services from external providers can or should be provided. In this regard, we can differentiate between three platform-specific parameters which we will address below.

Exclusivity

A key parameter in deciding the degree of openness is the agreement of exclusive rights. Their goal is to ensure that a specific complementary product is offered exclusively via the platform (Eisenmann and Wong 2004). For example, new versions of the popular computer game Grand Theft Auto GTA are first available only on Sony's Playstation platform. A second form of exclusivity agreement is category exclusivity. In this case, the platform operator and complementor agree that a certain type of application may only be offered by that particular complementor. Strategies such as these, which essentially reduce openness, make particular sense when one side needs to make high and specific investments, and is only prepared to do so when the other side ensures exclusive access to the distribution channel in return. This form of agreement can also be observed on the market for console games: Platform

operators commonly limit console games' access to the market—ensuring that, on one hand, the chosen games are of sufficiently high quality; and on the other, that they can command high licensing fees.

Reverse Compatibility

When a platform is further developed, new and/or additional functionality usually becomes available to the developers of complementary applications. An important decision is the required about reverse compatibility; in other words, whether to ensure that complementary extensions developed for an older version of the platform will also be able to run on newer versions. Platform providers need to weigh up whether to accept potentially higher costs and limitations in the further development of the platform, in order to make all existing complementary offerings available on new platform versions (Choi 1994). When partner applications are a central feature of the platform, ensuring reverse compatibility is essential. As proclaimed by its slogan, “There’s an app for everything”, Apple has made the availability of complementary applications a key competitive advantage of the iPhone. In consequence, the company needed to ensure reverse compatibility when developing the fourth generation iPhone. This is why the company deliberately opted for a new display resolution with exactly double the number of pixels, both vertically and horizontally. This step ensured that all applications developed for previous generations of iPhone will run on the new version without a hitch. If Apple had overlooked this, all developers of complementary applications would have been forced to provide a new version tailored to the new platform. In turn, this would have diminished the amount of vertical openness.

Integration of Complementary Applications into the Platform Core

Another method of determining the degree of openness is to integrate complementary applications previously developed by external partners into the core of the platform, in the course of further development. An example of this type of strategy is Microsoft’s Windows operating system: A multitude of standard applications, such as browsers, media players, and system utilities, were previously offered only by third parties as optional extras. Now, they are an integral part of current versions. As will be explained in the following section on the management of complementors, this step can be viewed as closing the platform, as these partners run the risk of competing with the operator (Yoffie and Kwak 2006). At the same time, there are reasons for this kind of strategy, such as reducing strategic dependence on particular providers, or the opportunity to realize economies of scale.

5.3.1.2 Horizontal Openness

A platform’s openness to other platforms or third parties is described as horizontal. This can be achieved via the following parameters (Eisenmann et al. 2009):

Interoperability with Other Platforms

The main method by which a platform can be opened horizontally is the provision of open interfaces and tools (Katz and Shapiro 1985). The Facebook Connect interface is an example of this way of creating interoperability: Once a user has registered, other platforms, such as Yahoo, can access their profile data. At the same time, Facebook even displays the user's activities on other platforms. As the Facebook platform had already reached a certain maturity by the time this converter was introduced (late 2008), this strategic opening provided an opportunity to grow user numbers beyond the core target group. Conversely, using Facebook Connect is also attractive for competing platforms, as it enables them to access the data of users already registered on Facebook.

Licensing Other Platform Operators

In the development stage of a platform, one-sided subsidies are often employed to prevent the "penguin effect" discussed in the context of network effect theory (see [Sect. 2.2.2.1](#)). In this stage, it often makes sense to license only one proprietary platform operator, to prevent freeloaders from taking advantage of one-sided subsidies (Eisenmann 2008). If the platform has reached a certain level of maturity; however, a strategic opening through the licensing of further operators can dramatically accelerate growth. This option is especially attractive when the additional operators apply their specific knowledge to provide innovative forms of the platform, which broadens the potential user base (Gawer and Cusumano 2002).

A successful example of this type of opening can be observed in the market for smartphone operating systems: The Android operating system is developed and provided by the Open Handset Alliance, led by Google. Android is licensed to numerous smartphone manufacturers, such as HTC, Motorola, Samsung, Dell, and Sony Ericsson, as an operating system for their devices. This strategy of licensing further providers (without relinquishing control over the development of the platform), seems to be a success: Android, at least for the time being, is recording strong growth rates, both in absolute terms and relative to its competitors.

Acquisition of Platform Sponsors

In addition, a platform can also be opened horizontally by taking on sponsors. In contrast to licensees, who base their specific extensions on the platform core, platform sponsors, are also involved in the ongoing technical development of the platform. We have already discussed the advantages of these types of development partnerships in [Sect. 3.1.1.2](#). On the other hand, opening the platform to additional sponsors also increases the complexity of coordination between the sponsors and increases the effort required to specify common standards (West 2006).

Compared to licensing additional operators, adding to the number of sponsors also involves a considerable risk: In the worst case scenario, political disagreements between the sponsors could delay or completely impede development. According to West (2003), this type of opening should be pursued if the original

platform operator's business model focused not on licensing the platform, but on selling complementary products or services. In fact, pushing this kind of business was a key factor in IBM's decision to transfer its rights from the Eclipse development platform to the open source community. We will delve deeper into this topic in [Chap. 7](#).

Another possible motivation for opening to new sponsors is when the platform is under considerable pressure from competing platforms. For example, in 1998, after losing the browser wars, Netscape disclosed the complete source code of its Netscape Communicator as part of the Mozilla Project, and opened itself to further sponsors. The result: the Mozilla Foundation produced the Firefox browser.

5.3.2 Management of Complementors

Senior management at software companies tend to focus on analyzing their own strengths and those of competitors, and frequently neglect to evaluate their supposed allies: providers of complementary applications. While platform operators are fully aware of their reliance on complementary functionality; especially those who run industry platforms, they often overestimate the extent to which their interests coincide. Even if, thanks to indirect network effects, both parties are equally eager for the platform to grow, their interests cease to accord when it comes to dividing up the profits.

For this reason, managing complementors is a key task for platform providers. In addition to an intensive analysis of complementors' business models, strategies, goals, skills, and motives, this also encompasses the selection of suitable paradigms for shaping the relationship between platform operators and complementors. Ney (2004) has devised two opposing paradigms, hard and soft power, which will be briefly described in the following section.

The most immediately obvious means for influencing complementors fall into the "hard power" category: By adopting a credibly threatening posture or dangling financial incentives, for example a share of revenue, platform operators hope to ensure that complementors toe the line. A real-life example of the use of hard power can be seen in Bill Gates' threat to cease the development of Microsoft Office for Apple's Mac OS, should Apple continue to refuse to integrate Microsoft's Internet Explorer in the Mac OS. These methods are generally underpinned by traditional sources of power, such as a large market share or exclusive control of a distribution channel. A platform operator can also exercise hard power and reduce complementors' independence by producing and providing strategically important complementary offerings itself. Smartphones are a case in point: Despite the trend towards coordinating a wide array of apps through marketplaces, vital core functionality, such as telephony or text messaging, remains integral to the platform. This strategy lets the platform operator realize economies of scale and generate additional earnings by selling complementary products. In addition, it can also be deployed to convey a clear message. However, especially when the

platform operator relies on the existence of a broad range of complementary offerings, this step can be counter productive: the message that the platform operator can and will encroach on the market and jeopardize complementors' sources of income may cause the latter to think twice about collaborating with this operator.

This reveals the disadvantages of using hard power: entering the market for complementary offerings incurs considerable ongoing costs. In addition, it prevents the development of a long-term relationship of trust with complementors. Furthermore, complementors will presumably try to avoid becoming too dependent on a particularly powerful platform operator, and may well bestow their long-term support on a competitor.

The alternative, namely exerting soft power, is generally a cheaper, and in the long term, more successful method of encouraging complementors to collaborate by pointing out shared objectives and opportunities. Concrete steps to achieve this include the pro-active communication of market data and plans for the future direction of the platform. Soft power can be wielded by proclaiming a common vision, which clearly highlights the advantages for complementors. One example of this, albeit from another industry, is Steve Jobs' efforts to integrate offerings from all the major music labels in his iTunes platform: In 2003, he articulated a compelling shared vision for the industry, and was able to persuade all labels to collaborate with him on terms that ensured the Apple iTunes Stores could offer attractive prices.

The disadvantages of this approach are that soft power is only successful as a long-term strategy, and an operator relying solely on soft power can be outflanked by a more aggressive competitor.

As shown above, both hard and soft power can be successful means to deal with complementors. Yoffie and Kwak (2006) have identified three factors that help to select the most suitable paradigm:

- **Strength and dominance:** The use of hard power in particular requires the deployment of considerable (generally financial) resources and a correspondingly strong position on the market. If a platform operator cannot meet these requirements, they are better advised to consider soft power. Small and seemingly weak companies can be especially attractive to external partners, as the latter do not have to fear that the operator will encroach on their territory.
- **Diversity of complementary offering:** If a platform operator is dependent on a large and diverse offering of complementary products, soft power is the better option, as this is the only way to ensure that the platform remains attractive to complementors over the long term.
- **Specific investments by complementors:** To integrate their offerings into the platform in the best possible way, complementors often need to make specific and irreversible investments. Where such investments are necessary, potential partners will try to guard against a breakdown in their relationship with the platform operator. As a result, it can be assumed that this type of trust is more likely to be built using soft power.

For platform operators, therefore, the question of hard or soft power is not an either-or decision, but rather a search for the ideal combination of the two. A middle path chosen by many operators is to restrict their exercise of hard power by producing only the most strategically significant complementary products themselves. Beyond that, they avoid intervening in the market for complementary offerings, or do so only in a very rudimentary way, for example in the form of quality controls. They will then use soft power to establish as stable and open a relationship with their partners as possible.

6.1 Overview

This chapter is devoted to cloud computing, a new form of IT service delivery via the Internet. The main focus will be on the software as a service (SaaS) concept, as this book primarily discusses issues relating to software rather than hardware. SaaS is regarded as a key trend—and figures on many IT decision-makers' agendas. SaaS involves providing a standard software solution to customers in the form of a service over the Internet. The SaaS provider is responsible for the operation and maintenance of the multitenant software. These providers do not charge license fees. Instead, users pay fees for the right to use software components and services. These are generally paid monthly, quarterly, or annually (for empirical findings on this subject, please refer to [Sect. 6.4.2](#)). In addition, software and service providers may leverage other revenue models, such as advertising or pay-per-use.

To come straight to the point: The idea behind SaaS is nothing new. Indeed, critics often gibe that it is “old wine in new bottles”. A similar approach, termed application service providing (ASP), was already being pursued in the 1990s (Günther et al. 2001). SaaS is nothing more than an extension of ASP that,—due mainly to the development and widespread adoption of innovative Internet technologies and standards—has a great deal of potential and opens up new possibilities for users and providers. To leverage SaaS solutions today, most users need only Internet access and a Web browser. Formerly, by contrast, taking advantage of ASP services necessitated high upfront investment and considerable expertise. For users, this means that switching to SaaS is generally simpler and, therefore, more cost-effective than it once was. Moreover, service-oriented architectures and open standards, such as Web service protocols, make it easier to integrate SaaS solutions with in-house systems and other services. However, to observe that SaaS is nothing revolutionary is not to infer that this concept will not continue to spread or that it is of little interest to providers and users.

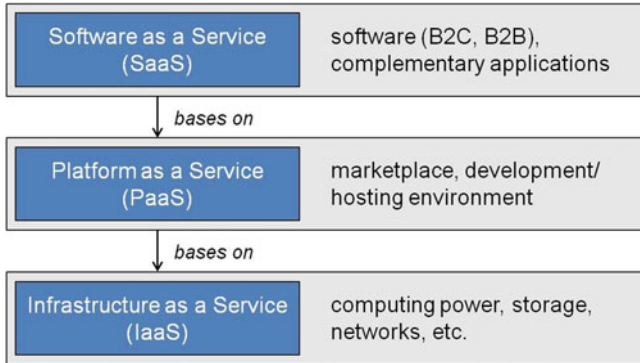


Fig. 6.1 Spectrum of cloud computing offerings (Vaquero et al. 2009)

In Sect. 6.2 we will begin by giving a general overview of cloud computing. Building on this, in Sect. 6.3 we will describe potential applications of SaaS and give examples of well-known SaaS products. Section 6.4 will offer empirical findings on the adoption of SaaS solutions from the user’s perspective, before discussing the provider’s viewpoint in Sect. 6.5. A key focus of the latter section will be on empirical findings regarding the business and pricing models of software providers.

6.2 Basic Principles of Cloud Computing

The basic idea behind cloud computing is that providers deliver standardized services to customers via the Internet. For users, this offers an opportunity to save costs and benefit from greater flexibility. Providers aim to make efficient use of resources and increase revenues through new business models.

As is commonly the case with new IT concepts and solutions, a multitude of definitions has sprung up. Here, we shall follow the definition offered by the national institute of standards and technology (NIST): “cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.” (cf. Zhang/Cheng/Boutaba; Mell/Grance).

Some examples are the virtual computing environment, Amazon Elastic Compute Cloud (EC2), Google Apps, and Microsoft Azure. Now, almost all major IT players, such as Dell, Hewlett Packard, and IBM, are offering cloud solutions. But even companies from other sectors, such as the financial industry and research, are considering putting their excess computing resources on the market.

Cloud services are normally categorized as follows: infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS) (Fig. 6.1).

IaaS solutions include computing power, storage, and networks. The customer manages these leased infrastructure components via a user interface. Two examples are EC2 and Amazon Simple Storage Service (S3). Amazon EC2 is a virtual computing environment. Users can lease preconfigured instances, each of which is effectively a virtual PC. The Small Instance includes a 1.7 GB main memory, a virtual core with an EC2 Compute Unit, and 160 GB instance storage (hard disk storage) based on a 32-bit platform. Amazon S3 is an online storage Web service. In conjunction with tools or further cloud services, S3 can be used as an external hard drive with unlimited scalability.

PaaS solutions encompass a variety of services that support the development and sale of software. The idea is that software developers—from individual programmers to leading vendors—develop complementary applications for the corresponding platforms. Apple’s App Store is an example from the business-to-consumer (B2C) space. In the business-to-business (B2B) space, there is the AppExchange platform provided by Salesforce.com, which enables users to develop and market CRM applications. More than 1,000 such applications are now available. Further examples of these types of platforms include Google App Engine and Microsoft Windows Azure (cf. Zhang/Cheng/Boutaba).

Software as a service involves the delivery of software to customers, who access it via a network. All the user needs is an Internet browser, which can be installed on a variety of devices, including mobile ones. We will discuss this subject in greater detail later in this chapter.

Providers commonly offer multiple service models. For instance, Salesforce.com provides its CRM system as a SaaS solution, while its development and deployment platform, Force.com, and sales platform, AppExchange, are PaaS solutions. Google also offers a platform in the shape of Google App Engine, and a SaaS solution, Google Apps.

Another classification is based on the operating model, i.e., public and private clouds. Public clouds are open systems that are shared by multiple customers. They usually allow customers to flexibly expand their usage as required.

In contrast, private clouds, also known as private managed clouds, are closed systems—for example, belonging to a particular company or provider. Being more limited physically, they are less flexible. On the other hand, the stored data is better protected against unauthorized access by third parties. When both types of cloud are used, this is termed a hybrid cloud. For example, hybrid solutions allow users to lease additional external resources to cope with peak loads.

As regards technology, the concepts of virtualization and multitenant architecture are pivotal to cloud computing. Virtualization comprises software and hardware-based techniques that add a layer of abstraction between the user’s applications and the provider’s physical resources. This can be illustrated by looking at data storage in the IaaS layer: Providers can make efficient use of their resources by distributing customers’ data to servers with free storage space. For customers, however, this means that they often have no idea where their data are physically stored.

Multitenant architecture is closely linked to virtualization. It means that rather than providing each customer with a dedicated technical infrastructure, all customers use the same platform.

Setting aside private cloud offerings, cloud computing is a form of outsourcing, as functionality and/or processes are outsourced to third parties (see also [Chap. 4](#)). As a result, the advantages and disadvantages of cloud computing are similar to those of outsourcing:

- **Concentration on core competencies:** By outsourcing tasks such as day-to-day application management or data storage, customers can free up resources for other activities, such as their core competencies.
- **Simplification and greater flexibility:** Cloud computing enables customers to streamline their IT management and make it more flexible. For example, it can be scaled up or down at short notice as required. Providers can add or remove storage and computing resources on demand.
- **Cost savings:** This is a key benefit of cloud computing. Fixed hardware and HR costs are converted into variable usage fees, avoiding costly capital expenditure. By exploiting economies of scale, providers are able to pass on attractive prices to customers.
- **Access to external knowledge and skills:** While cloud computing does enable customers to draw on external expertise, experience shows that outsourcing often entails the loss of in-house knowledge and skills. This is a drawback for one thing because it means the user organization is more heavily dependent on the outsourcing provider. If there are standards that make it easier to switch between providers, this becomes less of a problem.
- **Security:** Even major providers who work to the highest professional standards cannot always prevent system failures, as evinced by the server outages experienced by Amazon, Google, and Salesforce.com. The costs incurred by downtime vary widely, depending on the industry and the company affected. With online broker systems, for example, these costs are estimated at 6.5 million dollars an hour. On top of the direct costs of the outage, there are damages that are harder to quantify, such as loss of reputation and angry customers and suppliers.
- **Protection of customer data:** Another central issue for cloud computing is the protection of customer data from loss, hackers, or other threats. For example, a server fault at T-Mobile USA resulted in the deletion of data belonging to thousands of Sidekick users. The company was not able to restore all of the lost data. Of course, these types of slip ups are not the preserve of cloud providers alone—they also happen to customers themselves, as the recent incidents at games vendor Sony aptly demonstrate.
- **Lock-in:** The oft-times inadequate interoperability between different providers' offerings and with existing (legacy) systems poses another risk. As the standards for application programming interfaces are still incomplete, changing providers can prove to be a costly exercise for customers. As a result, this increases customers' dependency on the cloud provider. However, this lock-in

effect is not exclusive to cloud computing, either. High switching costs are associated with other types of standard software, such as ERP systems, which is why companies seldom change providers. Like ERP, cloud computing can also necessitate organization changes at the customer end. However, less scope for customization means that there is also less of a lock-in effect than with ERP or other standard solutions.

- Further risks: Measuring and monitoring the provider's performance can also be difficult. Another issue is how much the customer's IT organization will need to be restructured as a result of cloud computing.

This list is by no means exhaustive: Given that cloud computing is a relatively new area, further problem fields are sure to surface.

Simply put, the business model of cloud providers consists in providing customers with IT resources in return for a fee. That makes cloud computing particularly appealing to companies that already have extensive resources, such as Amazon and Google. The less use they make of these resources themselves (e.g., storage space, or processing power), the more the providers stand to profit. As a result, a cloud provider can realize economies of scale, delivering cost advantages in relation to IT infrastructure, hardware, software, and human resources.

For IaaS, customers are mostly charged a usage-based fee. Hourly rates can vary depending on the resource or service used and the user's location. For Amazon's Small Instance, these costs amount to a few cents per hour. Interestingly, Windows users tend to pay higher prices than Linux/Unix users. A fixed fee may also be offered, which reduces the variable costs per hour. This type of pricing is particularly attractive to heavy users. In addition, processing power and storage capacity may be auctioned. PaaS providers often make free tools available to developers, to entice them to produce applications for the platform. One such example is the Force.com development environment provided by Salesforce.com. Usage-independent fees are prevalent with SaaS, with the number of users being the main parameter. For instance, Google offers SMEs services like G-Mail, a calendar, Google Sites, word processing, and a spreadsheet program for 40 € per user per year. Customers of SAP Business ByDesign can choose between a number of versions priced between 19 and 199 € per user per month, depending on the scope of functionality, and include operation and maintenance (as at April 2011).

From a technical perspective, nothing about this modern version of outsourcing is revolutionary. Similarly, the principle of virtualization is not really all that new. However, cloud computing is interesting from an economic perspective, given that the costs are significantly lower than those of traditional outsourcing offerings. Moreover, thanks to open interfaces, the cloud can be accessed relatively quickly and simply, i.e., the barriers to entry are low. Competition between cloud providers is likely to increase dramatically, which will influence prices. As a result, cloud computing will have a considerable impact on users' IT. In turn, this could mean that cloud computing will effect lasting changes on the IT industry.

In the following sections, we will take a closer look at software as a service, the cloud computing layer most closely connected to software.

6.3 SaaS: Applications and Examples

SaaS is suitable for a variety of applications. However, this business model particularly lends itself to functions and processes that can be standardized to a high degree. This includes CRM software (see also the empirical findings in Sect. 6.3.2).

The following pages offer three SaaS case studies: salesforce.com, SAP Business ByDesign, and Google Apps.

Software as a Service at Salesforce.com

Salesforce.com is a US-based provider of on-demand business applications and operator of a cloud platform. As is typical for SaaS solutions, revenues are not generated by the sale of software licenses, but through subscription fees for the CRM system components used by the customers. These components, plus the entire infrastructure, support, and other services, are delivered over the Internet.

In addition to sales automation, Salesforce.com also offers solutions for marketing, partner management, content management, innovation management, knowledge management, and customer service. The CRM application is offered in five different versions (Contact Manager, Group Edition, Professional Edition, Enterprise Edition, and Unlimited Edition).

Since 2007, Salesforce.com has offered an on-demand platform for Web-based applications, Force.com. Users and developers can employ Salesforce's infrastructure to develop custom applications, which they can either use themselves or offer via the AppExchange marketplace.

Customers pay monthly fees for using the software components and services they choose. That saves them the—frequently high—upfront costs of acquiring software licenses and implementation. With SaaS, upgrading software also falls within the provider's remit. Salesforce.com's contracts have a minimum term (agreed by the parties) and if they are not terminated, are automatically extended for the same period of time. Fees are calculated according to the number of licensed users and the fixed monthly costs. The latter varies widely between versions (€ 4 for Contact Manager and € 270 for the Unlimited Edition per user and month).

Salesforce.com was founded in 1999 by Marc Benioff, a former Oracle executive. Right from the start, the company delivered its software over the Internet. By 2008, Salesforce.com had conquered 10 % of the market for CRM systems, notching up annual sales of \$ 1 billion. This earned it third place, behind only SAP and Oracle. Moreover, Salesforce.com has won a series of awards for its innovative products. It now has more than 67,000 customers with over 1.5 million users, and employs around 3,600 people.

SAP's Business ByDesign

SAP ERP is mostly deployed by companies with large workforces. To enable further growth, SAP has expanded its ERP product portfolio to include offerings for small and mid-sized enterprises (SMEs). Products aimed at this group include Business One (for SMEs with up to 100 employees), Business ByDesign (100–500 employees), and Business All-in-One (up to 2,500 employees). Of these, Business ByDesign is delivered exclusively as an on-demand business application (SaaS). As is commonly the case with SaaS, SAP is responsible for hardware, software, service, and support.

The core of Business ByDesign comprises eight modules: Executive Management, Financial Management, Customer Relationship, Supplier Relationship, Human Resources, Supply Chain, Compliance, and Project Management. Users can be authorized for specific functions or entire areas. Because the modules are integrated, users can perform analyses across multiple modules.

One potential advantage for users of SaaS makes itself felt when changes are made to financial regulations or other legislation: with conventional licensing models, users often have to acquire and/or install corresponding software updates. By contrast, in the ideal case, SaaS solutions are updated by the provider, without the customer having to do anything at all.

Unlike SAP ERP, Business ByDesign can only be customized by programming to a minor degree. It can only be tailored to users' needs by configuring the standard version. However, it is possible to add functionality via the SAP NetWeaver platform. The standard solution supports 7 of the 17 industries identified by SAP, including automotive, high tech, and electronics.

Customers pay for Business ByDesign based on the number of users. In contrast to other SaaS applications (for example salesforce.com), it is not possible to select only one module.

Office Software as a Service by Google: Google Apps

In addition to its well-known search engine, Google offers a variety of other Internet applications—including the Google Apps software package. This includes the Gmail e-mail service (which is called Google Mail in the UK and Germany, to avoid any conflict with existing names). The package also contains a calendar, word processing and spreadsheet programs, and an instant messenger. Google Apps is primarily aimed at enterprise. However, consumers can access all components of the package individually via Google's home page.

As usual with SaaS, neither the user interface, the program logic, nor the data are stored on the user's computer. To use the software, all customers need is an up-to-date Web browser.

The price list for Google Apps is short. The free Standard Edition is aimed at smaller organizations and offers only limited functionality. Google's ads are displayed to users, and the storage capacity per user account corresponds in size to a private Google Mail account. The Professional Edition offers greater storage capacity and support services in addition to availability guarantees and programming interfaces for integrating the software with existing IT environments.

According to Google, it has acquired around two million corporate customers, since launching the Professional Edition. Apart from Google itself, high-profile clients include Procter & Gamble and General Electric. Google also works with developing countries to increase the take-up of its offering. For example, the official Google Blog reported that the software is deployed by some 70,000 students at universities in Rwanda and Kenya.

With its Google Apps offering, Google has well-established desktop solutions such as Microsoft Office and Open Office in its sights. Similar SaaS products include IBM's Lotus Live and Microsoft's Office Web Apps. Google Apps may have a competitive disadvantage in its relatively late entry to the market, in light of lock-in situations on network effect markets (also see [Sect. 2.2](#)). However, the growing market for on-demand office solutions offers potential for Google.

Besides these high-profile examples, a multitude of other SaaS solutions are available. Software providers are also showing signs of interest in this business model.

6.4 SaaS from the User's Perspective: Opportunities and Risks

6.4.1 Background

Taking advantage of a SaaS offering essentially means outsourcing, i.e., contracting out functions or processes to third parties (Buxmann et al. 2008b). This means that some of the potential advantages and disadvantages of SaaS can be deduced from those of outsourcing. Against this background, we will now explore the opportunities and risks associated with SaaS. Based on prior research into conventional IT outsourcing (Earl 1996), the ASP market (Kern et al. 2002) and early findings on the adoption of SaaS (Benlian et al. 2009), five categories of opportunities and five of risks may be identified (Benlian et al. 2010). These are outlined in [Table 6.1](#).

We will now analyze, in greater detail, the opportunities and risks outlined in [Table 6.1](#).

A key advantage of SaaS solutions from the user's perspective is the opportunity to reduce costs and improve cash flow, given that software solutions no

Table 6.1 Opportunities and risks of SaaS from the user's perspective (based on Benlian and Hess 2010 with extensions by the authors)

Chancen		Risiken	
Category	Description	Category	Description
Cost and Cashflow benefits	Delivery of SaaS applications may lead to lower overall costs and improved cash flow	Financial risks	SaaS customers may end up paying more for application provision (e.g., due to Internet outages, increased customization costs, or price rises); risk of higher opportunity costs, as SaaS applications are generally less adaptable to company-specific requirements
Strategic and operational flexibility	SaaS customers may have greater scope to change provider (e.g., due to short notice periods for termination and reduced dependence)	Strategic risks	SaaS customers may lose business-critical resources or knowledge when outsourcing their application development and management
Improved quality	SaaS providers may be forced to deliver a continuously high quality of service, given that their customers are able to terminate at short notice	Operational risks	SaaS providers may not fulfill the Service Level Agreements in terms of availability, performance, and application interoperability
Access to specific resources	SaaS customers may benefit from the SaaS provider's resources, skills and technologies	Security risks	Business-critical data may be transferred to the SaaS provider, and/or mission-critical processes may be negatively impacted
Concentration on core competencies	SaaS customers may find it easier to concentrate on their core competencies if they outsource application development and management	Social risks	Outsourcing applications to a third party may invoke opposition from employees or lead to negative publicity

longer need to be installed on the company's servers. Nor is there any need for testing, development, or maintenance. Users also save upfront licensing costs.

SaaS involves regular, fixed costs for operations, support, and maintenance. For this reason, it is often discussed in terms of a rental or subscription model. Most providers do not charge extra for updating the software. However, in addition to the application rental costs, users also have to bear implementation costs—for example, for the technical and organizational integration of the SaaS solution. The integration with existing in-house systems can be particularly challenging. Having said that, in the conventional model, users not only have to bear implementation costs, they also have to pay upfront license fees (refer to [Sect. 3.3](#)). On top of this, there are annual support and maintenance fees. Users also face update costs roughly every 7–10 years.

As a general rule, implementation costs (hardware, software, business process applications, human resources), including licenses, are higher for a conventional standard software solution than for SaaS (Altmann et al. 2007, p. 40). One of the reasons for this is that the solution is not dependent on a specific operating system or platform, and so there are little or no additional IT costs. This means that SaaS solutions can be made available faster.

In addition, the implementation costs of SaaS solutions are often lower, because they normally offer less scope for customization than conventional standard software (Buxmann et al. 2008b). Of course, this also means that the cost benefit must be offset against a reduced ability to tailor the software to specific organizational requirements.

Another—much-vaunted—potential opportunity arising from SaaS is greater (strategic and operational) flexibility: Enterprises can simply change SaaS provider, for example if targets set in the contract are not met. Because the installation of hardware and software is the provider's responsibility, users tend to enjoy greater independence. In general, they need to make fewer investments in their own IT infrastructure and can terminate the contract ahead of time at relatively short notice. Assuming that the SaaS provider stores the user's data in an open data format, migrating the data is also relatively easy. However, as several real-life cases show, this purported flexibility is often no more than wishful thinking.

In the following, we scrutinize this potential advantage (i.e., greater flexibility in terms of choosing providers) a little more closely. As outlined in [Sect. 2.2.4](#), changing standard software solutions generally entails high switching costs for users. This is particularly true of ERP systems—which means that in real life, users rarely change providers. What exactly is the reason for these high switching costs? The most significant factor is not the licensing costs of an alternative software solution. Rather, it is because most ERP software reflects the user's business processes—and may have driven their re-engineering. As a result, switching the provider entails significant and costly operational changes. In principle, the same applies to SaaS. Once these solutions are integrated into the user's IT environments, a certain lock-in effect is unavoidable. The more the organization has invested in the integration, the greater the lock-in effect—and therefore the dependence on the provider. However, because SaaS solutions generally involve limited customization, open standards, and service-oriented architectures, there is less lock-in than with conventional standard software.

The opportunity to improve quality is seen as a consequence of SaaS' lower switching costs for user organizations: SaaS providers are therefore compelled to respond to the wishes and requirements of their customers. Another factor named as a source of better quality is SaaS providers' specialization in the delivery of the latest IT infrastructures, and their economies of scale. Quality is also improved through the prompt installation of updates, patches, and extensions. Moreover, providers are well placed to analyze customers' usage of their software. Providers see this as an opportunity to better understand their customers' needs and make their solutions more user-friendly. On the other hand, not all users are overjoyed at the fact that their activities can be logged and evaluated. Similarly, as we

discovered in many of our interviews with users, automatic updates are not necessarily regarded in a positive light, either. This applies especially to updates that affect navigation of the software as this may not meet with the employees' approval.

Access to specialized resources, skills, and technologies is often mentioned in the context of quality improvements. Because of their specialization, SaaS providers generally have the means and opportunities to invest in the newest generation of information technologies. In addition, the employees of SaaS providers can specialize exclusively in SaaS application provisioning and amass expert knowledge that ultimately benefits the end customer.

Finally, any discussion of the advantages of SaaS will include the standard argument that outsourcing software development, customization, and maintenance to a specialist third party enables companies to concentrate on their core business. This frees up resources in the IT department (both human and financial), which can then be devoted to strategic tasks. For example, routine support activities can be outsourced to the SaaS provider, allowing the in-house IT team to concentrate on strategic IT projects.

On the other side of the equation, there are financial, strategic, operational, security, and social risks to consider. In particular, financial risks are associated with hidden costs, which are common in outsourcing. Often, these cannot be estimated at the time the contract is concluded. One possible explanation for these hidden costs is that user organizations need to engage specialist system integrators—for example, to adapt the software to the company's specific needs or integrate the SaaS application with in-house applications. However, hidden (future) costs can also arise when SaaS providers raise their subscription prices, after companies have invested in tailoring the solution and have migrated their data. Providers may also ask for additional fees for alternative access channels to the SaaS solution (e.g., via mobile devices). Last but not least, the user organization may incur considerable expenses (e.g., loss of sales) resulting from system outages or poor performance (e.g., slow Internet connections).

Strategic risks result from the fact that a company outsourcing mission critical resources puts itself in a position of dependence, which could limit its freedom to act. For example, companies may no longer be able to respond flexibly to changes in their own corporate strategy, because they have lost the ability to tailor the software to their individual needs. In this scenario, the software could be customized by the SaaS provider or a system integrator, of course. In reality, however, this could not be implemented quickly enough to generate a competitive advantage.

Finally, social risks allude to the danger that employees (e.g., in IT or user departments, or employee representatives) will oppose the delivery of SaaS applications by a third party—or in other words, the outsourcing of (what are presumably) critical business functions. This could not only damage the company's reputation but also cause friction inside the company, negatively impacting on productivity. However, this risk is not peculiar to SaaS, and can result from many other types of organizational change.

Table 6.2 Breakdown of sample (Benlian et al. 2010)

Category	Percentage	Category	Percentage
Number of employees		Sales in millions of euros	
<10	27.3	<1	28.2
10–49	25.4	1–9	41.3
50–99	20.8	10–99	16.9
>99	26.5	>99	13.6
Use of SaaS applications (in years)		Position of respondent	
0 (non-customer)	59.4	CEO, CIO	24.9
>0 (currently SaaS customer)	40.6	Head of IT	62.5
I have been familiar with SaaS for ... years		Commercial manager	8.4
<2	17.3	Other and N/A	4.2
>2	82.7		

6.4.2 Empirical Study on Opportunities and Risks for SaaS Users

6.4.2.1 Data Set and Methodology

To investigate the opportunities and risks of SaaS from the user's perspective, we conducted an empirical study (Benlian et al. 2010). In July 2009, we took a random sample of 2,000 companies from the Hoppenstedt company database and sent them an electronic and a paper-based questionnaire. The questionnaire was aimed at heads of IT and CIOs/CTOs, who would have the background knowledge required to answer the questions. Of the 2,000 companies, 349 (of which 142 were SaaS customers and 207 non-customers) responded, returning a total of 922 completed questionnaires. On average, therefore, each company evaluated two to three application types used in their organizations with respect to the opportunities and risks of deploying SaaS (whether currently or potentially in future).

Non-customers were asked to evaluate how knowledgeable they were about SaaS, to ensure that they had indeed looked into the topic. More than 85 % of non-customers responded that they were familiar with SaaS solutions. Only 5 % indicated that while they understood the principles behind SaaS applications, they had not yet thought about implementing any.

The sample included companies from the following industries: Mechanical engineering/automotive, wholesalers and retailers, insurance/banks, telecommunications/information/media/entertainment, real estate and construction, logistics, public sector and healthcare, and service providers. Further, characteristics of the respondent companies are outlined in Table 6.2.

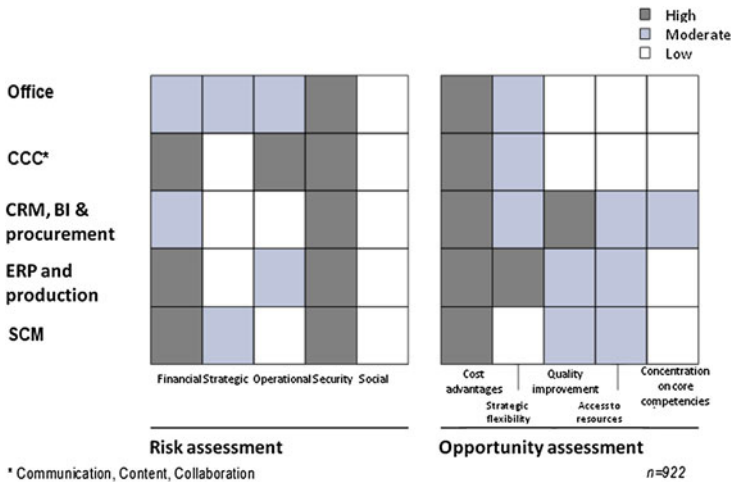


Fig. 6.2 Opportunity risk analysis for various application types (Benlian et al. 2010)

6.4.2.2 Findings

The participating companies were asked whether and to what extent they currently used SaaS applications, or were planning to do so in future. This question spanned the period between 2008 and 2012. Their current or planned deployment was measured in terms of the percentage of their IT budget spent on SaaS for the relevant type of application (e.g., ERP or CRM) (Benlian et al. 2010).

The results show that acceptance was greater for highly standardized applications such as CRM or office applications than for less standardized application systems. Expenditure on SaaS for highly standardized types of applications ranges between 8 and 14 % of the IT budgets for 2008 and 2009 and 23–35 % for 2010–2012.

Less standardized types of applications are much less likely to be delivered via SaaS, with just 0–3 % between 2008 and 2009 and between 4 and 11 % between 2010–2012. However, because of the low base effect, less standardized application systems will see higher growth rates in the near future. For ERP systems, we found an average growth of 54 %—and an impressive 95 % for SCM applications (Benlian et al. 2010).

In the next step, respondents were asked to gage the opportunities and risks of deploying SaaS. Enterprises cited short- to middle-term cost advantages as the greatest opportunity. Clearly, SaaS is regarded as a means to reduce costs. In addition, companies also regarded flexibility as a moderate to a major advantage. Furthermore, the respondents saw CRM, ERP, and SCM applications as offering moderate to good opportunities for improving quality and access to specialized resources and skills. Interestingly, SaaS is not seen as an opportunity to focus on the company’s core competencies.

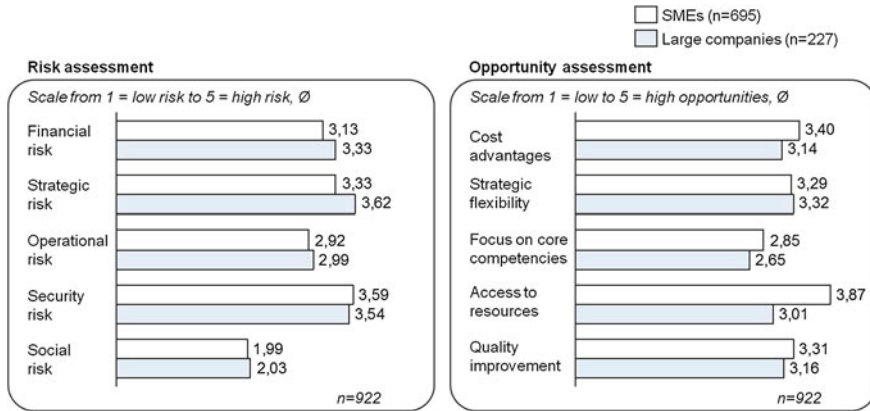


Fig. 6.3 Opportunity risk analysis: large companies compared to SMEs (Benlian et al. 2010)

According to the results of our study, security concerns emerged as the most significant form of risk, across all types of applications. In addition, respondents saw considerable financial risks in relation to CCC, ERP, and SCM applications. We can conclude from this that many companies fear that deploying SaaS applications will incur hidden costs that will only be revealed in the course of time. The companies in our sample did not see any social risks, either in terms of resistance from employees or damage to the company's reputation as a result of surrendering critical parts of the business.

Our results concerning the opportunities and risks of SaaS are visualized in the following diagram (Fig. 6.2).

If we compare the opportunity risk analysis for SMEs with that for larger companies, we find only small differences (see Fig. 6.3). For larger organizations, however, the strategic risk of losing business-critical skills to a SaaS provider is a particular concern. SMEs' greatest fear, on the other hand, is that the delivery of SaaS applications via an Internet interface could lead to loss of data and/or a connection failure.

In terms of opportunities, SMEs especially value the advantage of being able to draw on specialized resources, skills, and technologies that they themselves do not possess. They also saw greater cost advantages in SaaS than major enterprises did. By contrast, larger companies emphasized strategic flexibility and potential quality improvement over cost advantages. Neither SMEs nor large enterprises regard being better able to concentrate on their core competencies as a key advantage of SaaS.

A comparison of the opportunity and risk assessments of current SaaS customers and non-customers reveals some interesting differences. While on average, non-customers evaluate the risks of SaaS as being consistently high, SaaS customers tend to focus on the advantages (see Fig. 6.4).

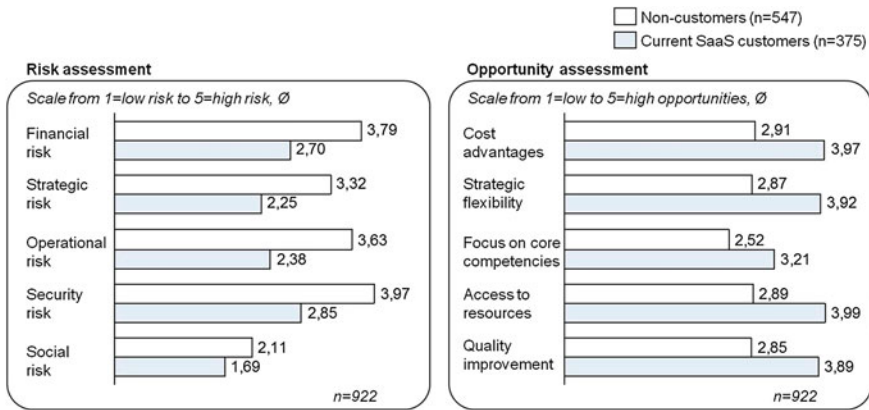


Fig. 6.4 Opportunity risk assessment: SaaS customers compared to non-customers (Benlian et al. 2010)

Analyzing the details, we see that security is a greater concern for non-customers. In addition, this group rates financial and operational risks as critical. Current SaaS customers regard security and financial risks in the same light. However, they see the risk of employee opposition to SaaS and its possible consequences (downsizing) as minimal.

6.5 SaaS from the Provider's Perspective: Pricing Strategies and Business Models

6.5.1 Basic Considerations

In the previous section, we could deduce some of the potential advantages and disadvantages of SaaS for users from those for outsourcing. We can do the same for providers. Assuming they have a large customer base, SaaS providers can exploit economies of scale and therefore cost advantages. This applies to IT infrastructure, hardware, software, and human resources. If the SaaS provider is also the vendor of the software in question, further savings can be made in development. Because the software is operated solely via a single platform, there is no need for costly modifications to make it run on different operating systems.

Moreover, SaaS may make it easier to offer customers a range of product versions. For example, different solutions for large organizations and mid-sized enterprises can be provided through the same platform.

Companies that offer SaaS are naturally subject to the risks that every provider of software faces. In addition, users' diminished dependence on a particular provider, as discussed in the previous section, can also pose a risk for SaaS providers.

SaaS providers can also learn from the results of the study presented above, specifically that non-customers are still apprehensive about deploying SaaS. Their

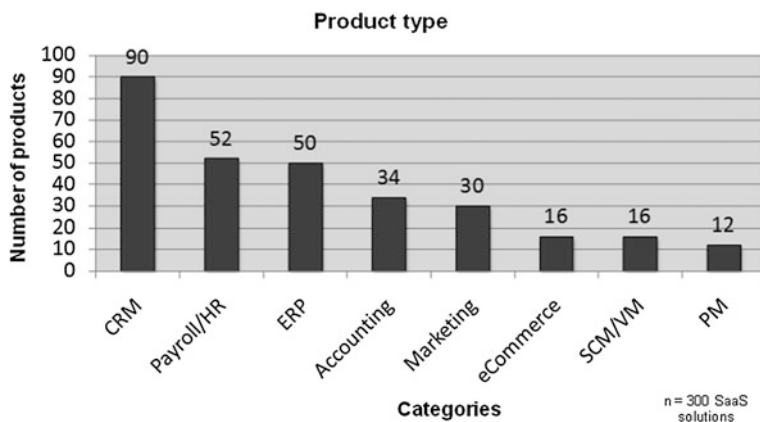


Fig. 6.5 Surveyed SaaS solutions by product type

main concerns relate to security and financial risks. Consequently, SaaS providers should improve their communications by focusing on the positive experiences of current customers to win over the rest. SaaS providers should also avoid tailoring their offerings solely to SMEs. Our results show that large user organizations appear to evaluate opportunities and risks much the same as SMEs, which suggests that they have a similar need for straightforward, affordable, on-demand software solutions.

In the following section, we will continue to examine SaaS from the provider's perspective. In particular, we will investigate the various pricing strategies pursued by software companies.

6.5.2 Empirical Study of SaaS Providers' Pricing Strategies and Business Models

Between February and May 2010, we conducted a content analysis on the websites of 259 US providers, to identify their pricing models for SaaS products. In addition, the study encompassed statistics about the provider (the size of the company and the year it was founded) and the product type of the SaaS solutions on offer.

To find SaaS providers for our study, we used the "Software-as-a-Service Showplace".¹ Currently, this Internet portal lists more than 1,300 SaaS providers. According to the portal's operators, the majority of these enterprises are US based. Providers can register themselves according to the type of application and industry.

The study's target group was providers with SaaS products for business customers. The sample was also limited to US-based providers. We cannot guarantee

¹ <http://www.saas-showplace.com>

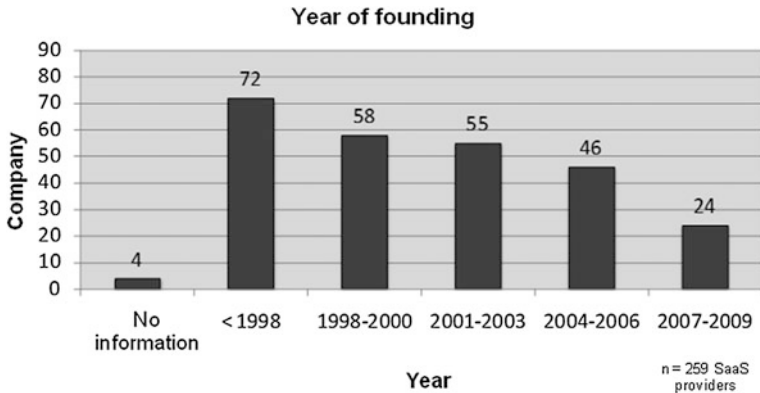


Fig. 6.6 Breakdown by year of founding

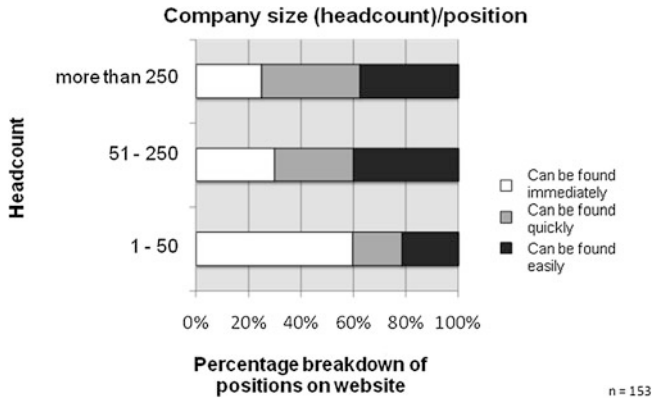


Fig. 6.7 Effort needed to find pricing information on SaaS providers’ websites in relation to company size

that this group is representative of US SaaS providers as a whole. However, as we included all providers in the B2B category, and other portals offered few additional providers, we shall assume that our sample can be regarded as characteristic of the industry as a whole. The portal’s list is regularly updated and expanded by the site’s operator, as well as through registration by SaaS providers.

Of our sample group of 259 companies, 56 % had up to 50 employees. A fourth of providers had between 51 and 250 staff. Only 13 % had a headcount exceeding 250. We were unable to establish the number of employees for the remaining 6 %.

Figure 6.5 shows a breakdown of the 300 SaaS solutions in our analysis (one company can offer multiple solutions) by product type.

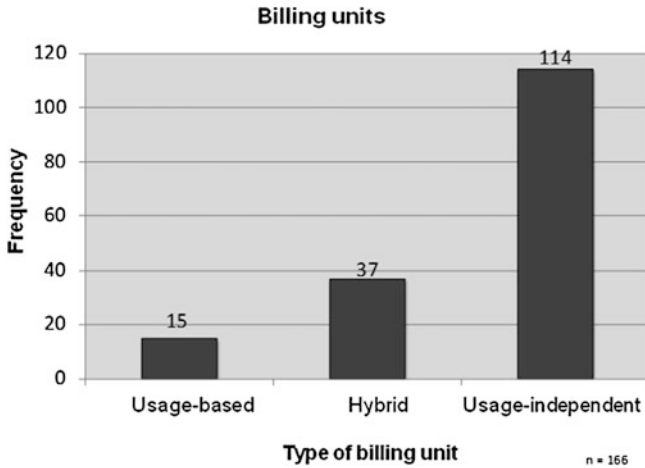


Fig. 6.8 Use of different types of pricing basis

As SaaS is a relatively new form of software delivery, we also looked at the year in which each provider in our sample was founded. The following diagram shows the results, divided into 3-year intervals (Fig. 6.6).

Our empirical investigation focused on the billing units used, in order to discover to what extent usage-based pricing models are employed, and by which providers. The availability of this information on providers' websites gave us an indication of the transparency of their pricing models. In most cases, providers' websites stated both the billing units and the corresponding prices.

We found details of billing units on the provider's website for 55 % of the 300 SaaS solutions in our analysis. For the other 45 %, no other information was available. The availability of this information is a good approximation for the transparency of pricing models for customers: The billing units and corresponding prices are the crucial information that customers need in order to understand the product's pricing model. We then investigated which providers published pricing information on their websites, and which did not.

More than 60 % of small SaaS providers (those with up to 50 employees) provided an above average amount of pricing information on their websites. In contrast, this applied to less than 40 % of the providers with a workforce of more than 250. A more detailed analysis of the smaller enterprises confirmed this correlation: Smaller SaaS providers tend to provide more pricing information on their websites than large providers.

With regard to the effort needed by customers to find pricing models on providers' websites, we found the following: (Fig. 6.7)

We found pricing information immediately on 60 % of smaller SaaS providers' websites, i.e., the pricing model was described on the home page, or there was a button labeled "price" or "pricing" that led to this information. The "Cannot be found easily" category included cases, where pricing information could not be

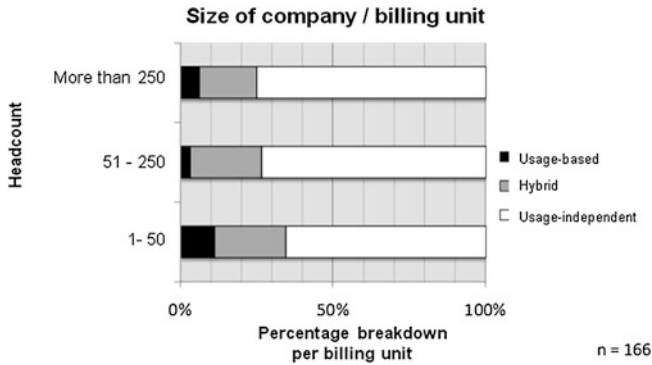


Fig. 6.9 Billing unit type by company size

found intuitively or quickly, for example, where it was hidden away in Terms and Conditions, FAQs, News, and other places.

Looking at the pricing basis for US providers' SaaS solutions, we can see that exclusively usage-based pricing models are rare. The majority, or 114 of the 166 SaaS solutions, are usage independent (Fig. 6.8).

On the basis of the available data, we are unable to satisfactorily answer the question of whether a particular type of billing unit is more common among particular types of providers: Usage-based pricing is seldom used by small, mid-sized, or large SaaS providers alike (Fig. 6.9).

Comparing the age of the SaaS provider with the billing unit employed also fails to bring any relationship to light. It is noticeable that, of the companies in our sample that were founded between 2007 and 2009, none of them employ exclusively usage-based pricing models.

Overall, we can conclude that usage-based pricing has not (yet) established itself as the primary model. Instead, user-based models continue to dominate. In the following section, we will look at this subject in greater depth. We will consider demand structures with usage-based and usage-independent pricing models through the lens of a case study.

6.5.3 Case Study to Compare Usage-Based and Usage-Independent Pricing Models

In the following section, we will further explore the issue of usage-based and usage-independent pricing of SaaS solutions using an example. In particular, we want to test the validity of the oft-cited claim that SaaS is especially suited to usage-based pricing (e.g., Kittlaus and Clough 2009, p. 59; Choudhary 2007).

Against this background, we carried out a case study involving a provider of statistical software for the B2B market. This software provider is planning to deliver an application via SaaS, which it currently offers as an on-premise version.

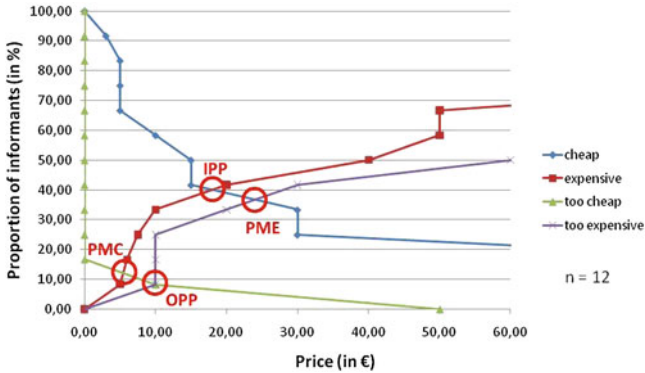


Fig. 6.10 Results of the PSM with usage-independent pricing (Lehmann et al. 2010)

In this context, the question arises of what pricing model should be employed for the new solution.

6.5.3.1 Data Set and Methodology

In February and March 2009, we conducted a telephone survey of the software provider’s customers. We used van Westendorp’s *Price Sensitivity Meter* or PSM (van Westendorp 1976) to compare customers’ willingness to pay for a SaaS solution when it comes with usage based as opposed to usage-independent pricing. Because of the small sample size and the opportunity we had to discover the reasons for these decisions by this means, we decided against other methodologies, such as conjoint analysis.

The van Westendorp method is a form of direct survey, whereby customers are asked four questions on a previously defined product. These questions ask respondents at what price they would consider the product “affordable”, “expensive”, “too expensive”, and “too cheap” (these being the method’s four price points). Individual participants’ answers are aggregated and presented as curves on a diagram. The intersections of these four curves define the region in which the price for the product should be located. The point of PSM is not to determine a concrete price-demand function. Rather, it is intended to identify an “acceptable price range” (Lock 1998, p. 507) for an innovative product when it is not yet apparent what the price should be.

The *sample* consisted of 28 of the software provider’s customers, who already use the intended SaaS product in its on-premise version. As a result, these customers are familiar with the application’s functionality and the costs of the current licensing model.

With respect to the size of the company, 38 % of our sample comprised small and mid-sized businesses whose annual sales are below 500 million €, while 62 % were large enterprises whose annual sales exceed 500 million €. The sample

companies were mainly automotive OEMs and suppliers in German-speaking countries. We will devote the following section to the survey's findings.

6.5.3.2 Findings

The van Westendorp method was implemented twice in this survey. First, "concurrent users" was chosen as the billing unit for the SaaS application's pricing model. A total of 12 out of 28 respondents, i.e., around 43 % percent, gave answers on all four price points. The results are illustrated in Fig. 6.10

As explained above, PSM provides a price range within which the price for the application should lie. This recommended price falls between the "point of marginal cheapness" (PMC) and the "point of marginal expensiveness" (PME)—in this case between € 5.46 and € 24.10 per month and concurrent user.

Our survey of the demand structure with usage-based pricing, which was conducted at the same time, had a much smaller response rate. One of the functions of the software is the production of statistical reports. Accordingly, "per completed report" was chosen as a usage-based billing unit for the software. Only 14 % of participants (four out of 28) gave concrete figures for all four price points.

However, the results do offer some insight into why usage-based pricing models play such a minor role. In addition to concrete price points, the telephone survey allowed respondents to give reasons for their inability to estimate a suitable price: It transpired that respondents were not actually sure how intensively they used the software they deployed. As this is what determines costs with usage-based pricing, these customers were unable to estimate their willingness to pay.

The choice of billing unit—such as the number of reports produced—turned out to be a further problem. Given the versatility of the software, and the many different ways it is employed by users, opinions varied as to what a suitable usage-based billing unit might be.

The results obtained from respondents' replies also contained some surprises. For example, two respondents' estimations of what was a "cheap" price varied by a factor of more than 1,000—which indicates that what customers would be willing to pay varied over a huge range. For the provider, this state of affairs is especially unwelcome. As we explained at some length in Sect. 3.3, to reach as many customers as possible and best exploit their willingness to pay, a homogeneous demand structure with a low variance is required.

Generally speaking, price discrimination is a good way of exploiting customers' different reservation prices. Given the magnitude of these differences, however, this may be difficult to implement. Furthermore, presumably due to their distribution over the Internet, the pricing of SaaS products tends to be more transparent than, for instance, on-premise software. This makes third degree price discrimination particularly difficult.

In conclusion, the study's findings do not support the oft-cited assertion that SaaS is highly suited to usage-based pricing. In most cases, providers employ

usage-independent billing units. For most customers, prices based on usage, such as per completed transaction, are optional.

However, it should be noted that this study of demand structures was conducted among potential customers of one particular SaaS offering. Comparable surveys relating to other types of SaaS products, such as ERP or CRM software, would be of interest, as would a general analysis of the relationship between the demand distribution and the form of pricing model offered.

7.1 Overview

Software providers in the narrower sense create software in order to generate license sales and in some cases, revenue from services. A different motivation lies behind open source projects. Software developers come together in an international community to pool their knowledge and jointly solve a problem. In this scenario, many developers invest their time, normally without being paid. But it does not follow that open source software (OSS) is irrelevant in an economic sense. In this chapter, we will explore the fundamental questions that OSS poses to the software industry and to users.

We will begin by briefly introducing the nature and features of open source software (OSS) and exploring the origins of the open source movement. We will then examine the development process in open source projects and how it differs from the process in a traditional software company. We will also consider what motivates developers to become involved in open source projects in the first place. Furthermore, we will look at the introduction of OSS from a user perspective. We will then provide an overview of commercial software providers' strategies—in terms of opposing and utilizing OSS. We will conclude with some thoughts and early empirical results on the use of open source business apps.

7.2 Features of Open Source Software

Free software has been around for a long time. Private users have frequently taken advantage of freeware, for example certain database systems or games. The methods of distribution have evolved over time: during the early years of the personal computer, freeware was exchanged via floppy disk, then via CDs, and nowadays almost exclusively over the Internet. Programmers have always been in the habit of sharing their source codes and programs to help and learn from one another.

In the 1970s, some companies began to sell only compiled software, and keep their source code under lock and key. A movement opposed to this practice evolved, and one of its pioneers was Richard Stallman. He began his academic career in 1971 at MIT's Artificial Intelligence Laboratory. The following quote from Stallman paints a picture of the culture that prevailed there at the time (Grassmuck 2004, p. 219):

“I had the good fortune in the 1970s to be a part of a community in which people shared software. We were developing software, and whenever somebody wrote an interesting program it would circulate around. You could run the program, add features, or just read the code and see how problems were solved. If you added features to the program then other people could use the improved version. So one person after another would work to improve the software and develop it further. You could always expect at least the passive cooperation of everybody else in this community. They might not be willing to drop their work and spend hours doing something for you, but whatever they had already done, if you could get some use out of it, you were welcome to do so.”

As the quote shows, free software is by no means a phenomenon of the 1990s. Many people viewed sharing software code and knowledge as a matter of course.

In the end, dissatisfaction with the functionality of a printer driver sparked the development of the open source movement (Grassmuck 2004, p. 222), although it was not yet known by that name. The Xerox network printers at MIT had no function for displaying printer status directly on the PC. Stallman wanted to write a function to make this possible, and embed it in the printer driver's source code. But the Xerox employee responsible for the code refused to release it, because he had signed an undertaking not to share it with any third parties.

This prompted Stallman to do two things: develop the driver himself, and set up the GNU project (GNU is a recursive acronym for GNU's Not Unix). In order to prevent others from making commercial use of his work, Stallman resigned from MIT. To make a living and ensure the continuation of the GNU project, he founded the free software foundation. This organization collected donations, charged fees for the distribution of GNU software on data media complete with manuals, but not for the software itself, and hired developers. A complete GNU/Linux operating system was created in the early 1990s by combining the components of the GNU project with a Linux kernel.

Free system components were not the only fruits of the GNU project: it also produced the GNU general public license (GPL), a special software license that has had a substantial influence on the free software and open source movements. The GPL grants users free access to the source code, the right to copy and share the software, freedom to modify the code, and permission to distribute the modified version, albeit under the same terms.

From an economic perspective, this last condition rules out the possibility of later changes to the software's property rights. Rather than, the software developer waving his intellectual property rights (which is standard practice with public

Table 7.1 Selected features of some open source licenses (adapted from Perens 1999, p. 186)

Type of license	GPL	LGPL	MPL	BSD license
Can be integrated into proprietary software and redistributed without an OS software license	No	Yes	Yes	Yes
Modifications to OS licensed source code can remain proprietary on distribution	No	No	No	Yes

domain software), the so-called copyleft principle is applied, to guarantee the software remains permanently free. In accordance with this principle, modified versions of the software have to be subject to the same license.

In addition to GPL, there are many other open source licenses, such as the less restrictive library/lesser general public license (LGPL, originally developed for libraries), the Berkeley software distribution-style license (BSD-style license) and the mozilla public license (MPL). Table 7.1 summarizes selected features of these licenses.

The term “open source” was not coined until 1998, when the open source initiative (OSI) was founded. Until that point, free software had been the standard name. But the change was more than just linguistic. Eric S. Raymond, known principally for his essay, “The Cathedral and the Bazaar” (Raymond 1999), in which he compared a centrally run software project to the construction of a cathedral and the decentralized organization of a project in the Linux community to a bazaar—was not the only one interested in a realignment of the free software movement. Software companies also displayed an interest. One reason for founding the OSI was Netscape’s announcement of its intention to publish its browser’s source code.

In summary, the aim of OSI’s founders was to set the free software movement on a new course. A key goal was to improve cooperation with software companies. Free software was renamed “open source” as a way to “market the free software concept to the people who wore suits” (Perens 1999, p. 173). Volker Grassmuck has commented that some developers probably feared that the word “free” could cause misunderstandings and could be interpreted as a communist “four-letter word” (Grassmuck 2004, p. 230).

Such considerations prompted more than just a change of name: the group wrote a definition of open source based on the work of Bruce Perens, the former project leader of Debian GNU/Linux. The definition included several criteria that must be met for software to be classed as open source. The criteria are listed in the box below.

The OSI open source definition

Open source does not just mean access to the source code. The distribution terms of open source software must comply with the following criteria:

1. Free redistribution

The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.

2. Source code

The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of downloading the source code, without charge, via the Internet. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms, such as the output of a preprocessor or translator, are not allowed.

3. Derived software

The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

4. Integrity of the author's source code

The license may restrict source code from being distributed in modified form *only* if the license allows the distribution of “patch files” with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.

5. No discrimination against persons or groups

The license must not discriminate against any person or group of persons.

6. No discrimination against fields of endeavor

The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.

7. Distribution of license

The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

8. *License must not be specific to a product*

The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.

9. *License must not restrict other software*

The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open source software.

© 2011 Open Source Initiative. Opensource.org site content is licensed under a Creative Commons Attribution Noncommercial No Derivatives License (creativecommons.org/licenses/by-nc-nd/2.5/legalcode).

This definition is not itself a license but a standard against which licenses are measured. In effect, the OSI assumes the role of a certification authority. So far, more than 60 licenses have been certified, including the GNU GPL, GNU LGPL, MPL, and the New BSD License. A current list can be found at <http://www.zopensource.org/licenses>.

Some of these licenses make it easier for software providers to privatize or commercialize OSS. This is why the OSI's open source definition is controversial and has sparked so many debates. For example, the first criterion on the above list does not rule out the use of open source code in commercial software packages. This is how a BSD license enabled Microsoft to integrate open source code into Windows. If the code had been subject to a GPL, Microsoft would not have been permitted to use it without Windows becoming free software (Grassmuck 2004, p. 299).

As shown above, the GPL also prevents the privatization of modified codes, whereas under the terms of Apache or BSD licenses this is permissible (Grassmuck 2004, p. 301). A modified version of OSS can, then, be sold without having to release the source code.

It quickly becomes apparent that many software companies can profit from the OSI's new approach. GPL can prove to be something of a hurdle to software projects (even noncommercial ones); for instance, if open source code has to be integrated into commercial products. In response, the LGPL was developed to make code sharing more attractive by allowing libraries to be integrated more easily. Integrating a GPL licensed library into a software program would mean that the entire software would have to be subject to the GPL. Since this is often not what programmers want, and in order to incentivize developers to use free libraries, the LGPL relaxes this condition (Grassmuck 2004, p. 290).

The less restrictive software licenses that simplify the privatization and commercialization of open source code are a double-edged sword. On the one hand, it could be argued that commercialization is not necessarily financially damaging to open source developers and that many software projects reap its benefits. On the other hand, many of those involved in open source projects are sure to regard this as an injustice and an attempt by companies to get something for nothing, which could affect their willingness to participate in these projects. We will come back to these incentives later, but first we will turn to the development principles that shape open source projects.

7.3 Open Source Projects: Principles and Motivation of Software Developers

The process by which, open source software is developed, is very different to software development in a commercial company. In the following section, we will examine the main differences and how they affect the structure of a project and the motivation of those involved in it.

7.3.1 Organizational Structures and Processes in Open Source Projects

An open source project usually comes about when somebody would like to solve a problem. In the previous section, we described how Stallman's dissatisfaction with a Xerox printer driver was the starting point for the GNU project. Another oft-cited example is the development of the LINUX operating system. Linus Torvalds wanted to run a Unix operating system on his 386 PC. Finding nothing suitable, he began to develop his own and published his source code on the Internet. As we all know, the project met with keen interest and a number of software developers became involved in developing it. Even Torvalds was surprised by this turn of events and has repeatedly stressed that he had never dreamed that it would be so successful.

The development of OSS is an evolutionary and distributed process. Raymond evocatively represents the development process as a bazaar and contrasts this with the so-called cathedral model, synonymous in his eyes with conventional software development (Raymond 1999). If commercial software development is viewed in the context of early models from the field of software engineering, then the two approaches differ widely. However, concepts of evolutionary and of agile software development are similar to elements of the open source movement's approach (Sharma et al. 2002).

Once a project has been set up, its success depends on the prompt establishment of a community around the software. It is always helpful if some modules are already available for testing and execution.

The development team's decision-making structure and the composition of the team itself are centrally important. Large-scale open source projects usually have a

coreteam composed of the developers who have been working on the project the longest, or who have contributed a large quantity of code. The core team of the Apache web server project comprised 22 programmers from six countries (Grassmuck 2004, p. 237).

In small-scale open source projects, the founder usually assumes the role of “maintainer”. He assumes responsibility for project coordination and quality control. Developers with a good track record often act as maintainers in large projects, where a two-tiered system lets them coordinate module development and have a say in decisions on general principle in the overall project. It is quite often the case that the founding member has a special role in the core team. The Linux project, for instance, had a team of five or six developers who tested and selected incoming source code before passing it on to Torvalds, who would make the final decision (Dietrich 1999). In contrast, the Apache project’s core team takes a democratic approach to decision making. The decision to integrate a module or not may be decided, for example, via mailing list.

The maintainers and software developers who contribute a great deal of code to the project are supported by a host of other people, who test the software, write the documentation, and provide localizations. It is often difficult to find qualified software developers to perform these supporting roles. Most developers regard documentation as boring, and in any case it is not the best way to boost one’s reputation.

Open source projects tend not to be wound up like conventional ones. Instead, development work may be discontinued once the user is satisfied with the solution, or if either the maintainer or key developers have lost interest in the project. If a maintainer becomes inactive, the development community can appoint a new one. Occasionally, a project splits, or “forks”. Forking happens when the core team can no longer agree on questions of principle.

Internet-based source code management systems such as concurrent versions system (CVS) serve as an important function in open source projects. Programmers download the latest version of a module from the CVS, work on it, test it with their own development tools, and copy the results back to the CVS repository. If several developers have been working on a file at the same time, their changes are ideally merged into a new file in the repository. If this causes conflicts that cannot be resolved automatically, developers must come to an agreement amongst themselves. At regular intervals, the core team flags certain branches of the source tree in the repository as a new release.

7.3.2 Contributor Motivation

Open source projects are based on the joint work of a frequently global community of software developers. They participate on a voluntary basis and the majority are not paid for their work. Their involvement implies acceptance of opportunity costs, from sacrificing leisure time, to passing up alternative paid positions, to neglecting their day job. This is particularly relevant to the core team members in open source projects.

This raises the question of what motivates developers to participate in open source projects. Some authors try to explain the phenomenon through the personal gains a developer can make with his contribution (Lerner and Tirole 2002). Other studies assume that most programmers are driven by intrinsic motivations (Kollock 1999). Against this backdrop, Franck (2003) distinguishes between

- Rent-seekers and
- Donators.

Rent seekers behave like a conventional “homo economicus”. Rent-seeking developers are looking for benefits beyond regular pay checks. Empirical studies have repeatedly shown that contributors hope to boost their reputation on the job or capital market, expect to improve their know-how, or assume that their activities will help them in their everyday work (Lerner and Tirole 2002). Hence, these developers do not become involved unless they expect a pay-off –in other words, they are seeking “rent.”

Conventional rational motives alone cannot account for the phenomenon of people contributing to open source projects. Time and again, empirical studies have shown that developers hope to get enjoyment and entertainment out of their participation in open source projects, as well as advancing the open source movement. Furthermore, many open source developers are pursuing other goals, such as freedom of information. And very often, their activities are actually an attempt to break market leader Microsoft’s virtual monopoly. Franck terms people for whom this is a top priority “donators” (Franck 2003). They believe that they are investing their time in something worthwhile. For “donators”, open source projects subject to a GPL, or another similar license, are attractive, because they have no reason to fear that their “donation” will be commercialized.

Successful open source projects often manage to create a governance structure that appeals to both rent seekers and donators. The open nature of source code gives rent seekers the chance to improve their reputation, as their contributions are visible and verifiable. The more rent seekers involved, the more appealing an open source project is to donators because the chances of success are higher. This avoids the conflict that is often evident between rent seekers and donators.

However, gone are the days when the teams for most open source projects comprised unpaid hobbyists (Brügge et al. 2004, p. 101 f). Instead, there are a number of open source projects in which salaried programmers and other specialists work alongside each other. Almost 30 % of the developers at open source platform Sourceforge.net, for example, are paid for their work. It is rare for an open source project to be initiated by a commercial enterprise. The Apache project, for instance, was begun by employees who were responsible for their companies’ Web servers. But such a situation does not rule out rent seekers and donators working together. In fact, it may even encourage cooperation, especially in the case of projects subject to a GPL license or similar.

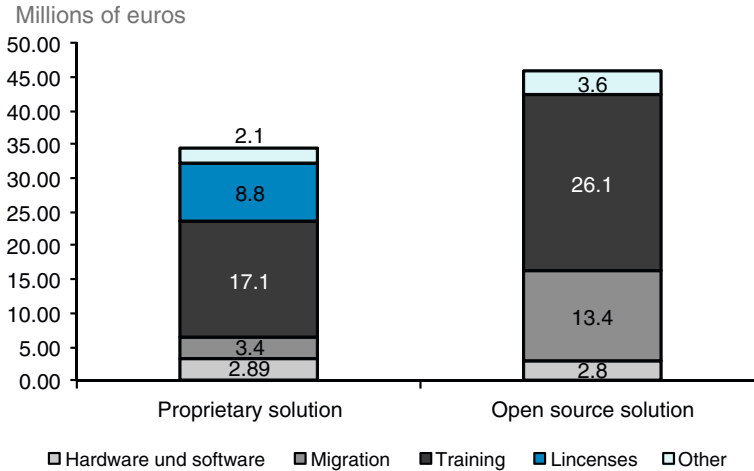


Fig. 7.1 Short-term cost comparison for the city of Munich

7.4 Open Source Software: The User Perspective

OSS has a particularly appealing benefit for users: it is free. This means that users save licensing costs of standard software, and expenses associated with company-specific solutions. But licensing costs are of course not the only relevant parameters that must be considered when choosing which software to deploy.

As an example, let us take a look at the decision by local government in Munich, Germany to replace Windows XP and Office solutions with Linux and open source software. This example was chosen, because the deployment rate of OSS in the public sector is particularly high. The study was carried out by consulting company Unilog in 2002 and considered the direct costs for the organization's 14,700 desktops. It concluded that migrating to an OSS solution is not the best option—at least not in the short term. It should be noted that the study focused only on costs. Although the city's licensing costs would be eliminated, any savings would be wiped out due to significantly higher migration and training expenses. Figure 7.1 shows a breakdown of costs (in millions of euros) for both options.

Upon considering the results of the study, the city of Munich still decided to migrate to the open source solution. It explained its decision by stating that it hoped to significantly reduce its dependence on Microsoft, and that the two solutions were identical in terms of their capabilities and standalone utility.

It is still unclear whether OSS can generally be regarded as cost-effective. Studies on this topic have reached varying conclusions (Brügge et al. 2004). For example, a Berlecon Research study (2002) found that, in addition to eliminating licensing costs, open source solutions have lower implementation and administration costs, and are more stable. But the latter two benefits were not demonstrated in the case of the Munich local government. Such great variance can be partly due to the fact that

differentiating between costs is far from simple in practice. In addition, the debate and the studies on this topic are often ideologically biased.

7.5 Commercial Software Vendors' Involvement

In a previous section, we examined why developers and other software specialists participate in open source projects. In this section, we will take a look at what motivates enterprises to support these projects. There are three main reasons (Hecker 1999, Raymond 1999, Henkel 2004):

- Supporting sales of complementary products and services,
- Integrating OSS into own products and
- Reduction of market power of competitors' proprietary software.

The reason most commonly cited in the literature is the possibility of selling complementary products and services. This is a follow-the-free strategy, as described in [Sect. 3.3.2.7](#). To put it another way: OSS creates additional demand on the user side. By meeting this demand, companies can create and leverage indirect network effects. Complementary add-ons can include hardware, software solutions, and services such as training. Linux distributor SuSe's business model is just one example.

Linux distributor SuSe

Linux is a free operating system that supports multitasking and multi-using. The system is now being deployed in many different areas (desktops, servers, mobile telephones, routers, etc.). Ready-made software packages, called distributions, are usually used. SuSe was the first company in Germany to successfully sell a Linux distribution on a broad scale. And it was one of the world's first companies to base its business model on Linux. SuSe penetrated the market with the first German-language installation program for Linux.

Background information

Roland Dyroff, Burchard Steinbild, Hubert Mantel, and Thomas Fehr founded Software und Systementwicklungsgesellschaft mbH (literally: "Software and System Development Company") in 1992. Their first product was merely an enhanced version of an existing Linux distribution; but in 1996, the company introduced the first distribution that it developed itself. The company's headquarters were initially in Fürth, Germany, before it was moved to Nürnberg, Germany in 1998. In 1997, SuSe opened an office in Oakland, USA. Six additional sites in Germany and three international ones (Italy, Czech Republic, UK) followed. In 2004, SuSe was acquired by American company Novell for 210 million dollars. Novell also assumed

responsibility for all of Suse's employees around the world, around 380 at the time. That year, SuSe generated revenues of 37 million euros. Today, Novell is one of the world's leading vendors of complete Linux solutions. A key success factor is its numerous strategic partnerships with companies including SAP, Oracle Intel, and IBM.

Service portfolio

Novell's product range includes offerings for both business and private users. The SuSe Linux Enterprise 11 platform is an end-to-end solution for enterprises. It comprises a database, server, desktop, and hardware management. OpenSuSe, an open source solution, is the company's offering for private users. Originally called SuSe-Linux, it was renamed to openSuSe in December 2006 to differentiate it from nonfree products.

Novell's main source of revenues is the provision of services. This is why its portfolio includes a comprehensive range of support, consulting, and training offerings, the most important being its certified courses for enterprises. To underline its service centricity, the company offers numerous free services such as a discussion forum, databases, and documentation that help users find answers to their questions.

Sources www.novell.com, www.opensuse.org

Companies quite often incorporate OSS in their products and solutions. Therefore, it makes sense for them to support their development financially. Integrating OSS in embedded systems is an example. This approach is characterized by a dedicated connection between hardware and software components that can only be used for one specific purpose. Machine control is a typical example. These systems often leverage Linux and other OSS. A high-profile example of a device featuring an embedded system of this kind is TiVO, a digital video recorder. If the all-but-invisible software used in a product is licensed under GPL, the provider can only generate revenues by selling (specialized) hardware. At its core, this is another example of a complementary add-on that we discussed above.

A third motivating factor for software companies to actively engage in open source projects and provide manpower for them is limiting competitors' market power. IBM's involvement in the development of Linux is just one example. By doing this, it succeeded in checking Microsoft's dominance in PC-based operating systems—at least to a certain extent. IBM's considerable investment in Linux gained it one significant benefit: reduced dependency on Microsoft as an operating system vendor.

As an alternative to paying employees for their involvement in open source projects, companies can publish the code of software developed in-house and donate it to the OSS community. IBM is one of the enterprises pursuing this strategy. Other well-known examples are Mozilla and OpenOffice, and Sun Microsystems' decision to turn over Java and JDK to the open source community.

7.6 Open Source ERP Systems

Today, when we talk about OSS, it is generally the well-known and successful projects, such as Linux, that spring to mind. And no wonder, more than 30 million copies of this operating system have been deployed worldwide. This corresponds to a 20–25 % market share for server operating systems. Other success stories include Apache Webserver, the Eclipse development environment, and the MySQL database.

A closer look at the above contenders shows that OSS has predominantly taken root at the lower level of the “software stack”, where it has become the standard. This leads to the question of whether and to what extent the OS paradigm is also suitable for ERP systems and whether it could compete with established software providers, such as SAP and Oracle, in this market segment.

One of the key issues is whether the ERP segment is alluring enough to attract many OS developers and whether they possess the necessary knowledge of business. A study of the largest repository for OS software, SourceForge.net, clearly demonstrates that developers are definitely interested in business applications. There are over 630 ERP-related projects on the platform. A notable example of how OS software can become established beyond the infrastructure level is the customer relationship management software, SugarCRM (Sterne and Herring 2006). Open source ERP systems have not been very prominent up to now.

A selection of open source and ERP production solutions is shown in the following table (Buxmann and Matz 2009). The table depicts how each of the responsible enterprises is organized, and describes the features of their individual software projects and characteristics of their business models. All of the offerings are international, multilingual projects. OS projects, such as aforementioned SugarCRM, or GnuCash, which supports financial accounting, are not included. These systems only support single functions of an ERP system (Table 7.2).

There are marked differences between the providers’ business models: The development of systems, such as ADempiere and webERP, is driven primarily by a committed community of private developers. The full functionality of the software is available under a GPL license, and paid services are marketed exclusively by third-party providers. Other offerings are more commercial in nature. Their business model is similar to traditional software providers. The only difference of note is that, in addition to the commercial software packages, they also provide a version with an OS license, which often only offers basic functionality. Both the provider and the customer can reap benefits from this dual licensing model (Mundhenke 2007, pp. 130–131; Hecker 1999, p. 49). Customers have the opportunity to test and use the free version without restrictions. And, if necessary, they can switch to the proprietary versions, which allow them to leverage richer functionality and services. This way, the provider can simultaneously address different customer groups and expand the installed base. Furthermore, many providers hope that releasing a software version under an OS license will have a positive effect in terms of marketing.

Table 7.2 An overview of open source ERP systems (Buxmann and Matz 2009) [Die Zeilen "Letzte Version" bittewiehierweglassen]

Provider	Adempiere	Compiere	ERP5 express	Openbravo	OpenERP
Provider/trademark owner	ADempiere	Compiere, Inc.	Nexedi SA	Openbravo, S.L.	Tiny
Location	USA	USA	France	Spain	Belgium
Website	adempiere.com	compiere.com	erp5.org	openbravo.com	openerp.com
Software project					
Project began	2006	1999	2007	2006	2005
Type	Web application and rich client	Rich client (Web interface is paid software)	Web application	Web application	Web application and rich client
Registered developers at SourceForge	89 (including 9 admins)	75 (including 2 admins)	Not registered	81 (including 15 admins)	Not registered
Programming language	Java	Java, JavaScript	Python	Java, JavaScript	Java, Python
Supported platforms (server-side)	Platform-independent	Platform-independent	Linux, MacOSX, Unix, Windows	BSD, Linux, Solaris, Windows	Linux, MacOSX, Unix, Windows
Supported databases	Oracle, Postgres	Oracle, Postgres	DB2, MySQL, Oracle, Postgres	Oracle, Postgres	Postgres
Business model					
Target group	No information	For organizations of all sizes	For organizations of all sizes	For organizations of all sizes	No information

(continued)

Table 7.2 (continued)

	Adempiere	Compiere	ERP5 express	Openbravo	OpenERP
License	GPL	GPL for Community and standard edition and a commercial license for the professional edition	GPL	Openbravo Public License (based on Mozilla Public License)	GPL
Price discrimination, software	None	Community, standard, and professional edition	None	None	Module-dependent pricing
Price discrimination, services	None	Community, standard, and professional edition	Community version, starter, premium, and elite packs	Community edition, SMB network, Basic network, OEM network	Module-dependent pricing
Provider	OpenTaps	xTuple ERP	SQL-Ledger	LX-Office	webERP
Provider/trademark owner	Open source strategies, Inc.	xTuple	DWS Systems, Inc.	Lx-System-Holger Lindemann, and LINET services GbR	Administrator: Phil Daintree
Location	USA	USA	Canada	Germany	New Zealand
Website	opentaps.org	xtuple.com	sql-ledger.org	lx-office.org	weberp.org
Software project					

(continued)

Table 7.2 (continued)

	OpenTaps	xTuple ERP	SQL-Ledger	LX-Office	webERP
Project began	2005	2002	2000	2004	2003
Type	Web application	Rich client	Web application	Web application	Web application
Registered developers at SourceForge	38 (including 1 admin)	39 (including 9 admins)	Not registered	4 (including 3 admins)	9 (including 2 admins)
Programming language	Java	C ++, JavaScript	Perl	Perl, PHP	PHP
Supported platforms (server-side)	Linux, MacOSX, Unix, Windows	Linux, MacOSX, Windows	Platform-independent	Linux, Unix	Platform-independent
Supported databases	MySQL, Postgres	Postgres	Postgres	Postgres	MySQL
Business model					
Target group	No information	SMEs	No information	No information	Small companies
License	Honest public license (based on GPL) and commercial licenses	PostBooks edition under common public attribution license 1.0 and standard and open manufacturing edition under a commercial license	GPL	Artistic License, GPL and LGPL	GPL

(continued)

Table 7.2 (continued)

	OpenTaps	xTuple ERP	SQL-Ledger	LX-Office	webERP
Price discrimination, software	None	PostBooks, standard and open manufacturing edition	None	None	None
Price discrimination, services	The commercial license provides enhanced support options	None	By user group: user, tech, dev	None	None

An empirical study of development activities and forum postings in the SourceForge platform (www.sourceforge.org) offers insight into whether the development of open source ERP software can be driven by a dedicated community of private developers. SourceForge provides developers and users with various tools for communication and software development free of charge. Registered members can take part in projects and use or add to the community's resource pool. Each SourceForge project generally has several forums. Participants can start threads, generally to ask a question, to which registered SourceForge users can then post replies. According to SourceForge, the platform currently hosts over 23,000 OS projects with over 2 million registered users. All projects whose software is governed by an OSI-recognized license can be registered. In collaboration with the University of Notre Dame, Indiana (USA), SourceForge provides data from its project database for research purposes (cf. van Antwerp and Madey 2008). This enables an analysis of the projects above and beyond what the platform's standard statistics tell us.

We will begin by taking a look at the ranking of the most active projects on SourceForge. A SourceForge project's activity level is derived from the number of visitors to the project site, development activity, and communications, for example within the forums. The standard ranking is based on a cumulative analysis of all data since the start of the project.

In this ranking, there are no ERP projects in the top 20 and only six in the top 1,000. However, recent rankings, which only cover the last 7 days, for example, paint a different picture. For the last 2 years or so (as of July 2009) ERP projects have figured high up on the list. For example, the Openbravo project is regularly among the top ranked, often landing in first place. Of course, these rankings say nothing about the quality of the projects' contributions. But they do indicate that, in principle, the OS model appears to be suitable for ERP software, too.

For a more indepth comparison of ERP and other OS projects, we will now examine the development and communication activities in greater detail.

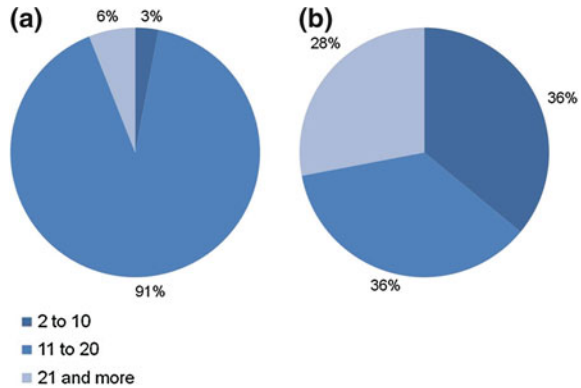
Our sample for this purpose comprises projects that meet the following criteria:

- at least two registered developers are involved,
- the project has existed for at least 1 year and
- the forums contain at least one posting.

These criteria filter out the very newest and the less active projects. They enabled us to identify 208 projects under the aegis of the ERP group. We then selected 208 projects unrelated to ERP at random.

We began the analysis by comparing the size of the communities. Leaving aside projects with only one registered participant, ERP projects have an average of 5.7 registered participants. As Fig. 7.2a shows, between 2 and 10 participants are registered in over 90 % of these projects. About 9 % of the projects have 11 or more contributors. The largest project (Openbravo) has 77.

Fig. 7.2 **a** Number of participants in ERP projects; **b** Number of participants in the control group (Buxmann and Matz 2009)



In contrast, the average number of participants in the other, randomly selected projects are 26.7, with a maximum of 391. Figure 7.2b shows that within the projects a similar number of participants are registered in each of the three categories—2 to 10, 11 to 20 and 21 and more. In conclusion, there are significantly fewer users and developers registered in the ERP projects than in the control group.

Next, we will examine and compare how participants communicate in the different types of OS project. We will begin by looking at the forums in ERP projects. Our findings show that around 80 % of users start threads, while some 48 % post replies on the various topics. At least 32 % of threads receive zero replies.

We notice some surprising similarities when we compare these findings to those of the control group, where the proportion of participants who start thread discussions is also 80 %. At the same time, 35 % of participants post replies, which is around 12 % points less than in the ERP project forums. With respect to randomly selected projects, around 72 % of their forum postings stay unanswered. It is hard to tell whether a posting has been answered satisfactorily or whether it is a follow-up question by the thread starter or merely fleshes out the original question. In order to filter out follow-up postings of this kind, we only included postings by users other than the thread starter. Our findings show that less than one-third of users only post replies in other people's threads, but do not start any of their own. Figure 7.3 gives the breakdown.

In summary, we found that users and developers in ERP project forums tended to be more active. In the group of randomly selected projects; for instance, the proportion of forum postings with zero replies was more than twice as large.

We will now examine whether there are any differences with respect to response times—the time lapse between the posting of a question and the first answer (Lee et al. 2009, p. 431). The following chart gives a comparison between the response times for ERP projects and those of other projects. Over 60 % of threads received a reply within 1 day, and 83 % after 1 week. There were no significant differences between the ERP projects and other projects in this respect (Fig. 7.4).

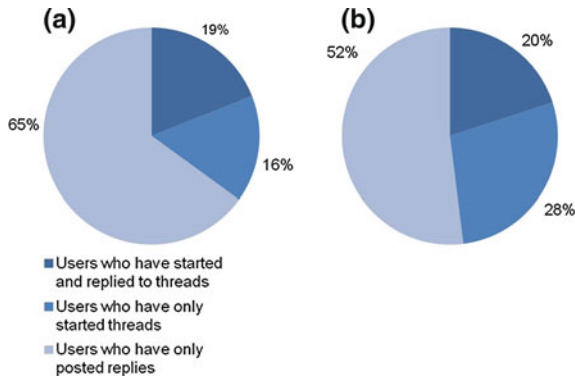


Fig. 7.3 a User activity in ERP project forums; b User activity in control-group forums (Buxmann and Matz 2009)

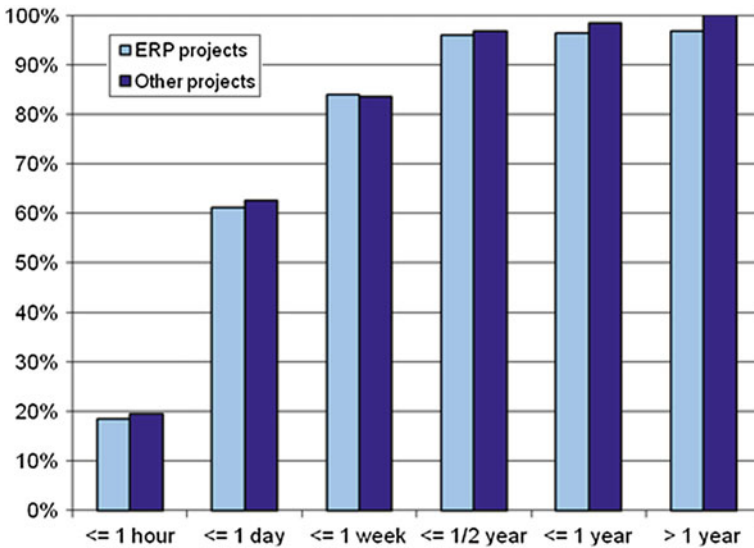


Fig. 7.4 Cumulated response times in ERP project forums and others (Buxmann and Matz 2009)

In conclusion, the OS scene plainly does now offer alternatives to proprietary ERP systems. Our study shows that although there tend to be fewer developers involved in ERP projects, these communicate with each other more intensively, responding to forum threads more willingly and more frequently. It is also noticeable that, ERP projects are increasingly among the most active projects in the SourceForgedatabase recently.

It must be emphasized that these findings say nothing about the quality of work; for example, how useful the posted responses actually were, or how much effort programmers put into the development process. Furthermore, the data we used did not allow us to differentiate between private and paid programmers. Many software and IT enterprises employ large numbers of programmers who spend all or most of their time working on particular OS projects. As a result, the findings of this study may only be interpreted as showing that the OS model can be successful in the ERP space. However, they do not allow us to estimate the market potential of ERP applications. On the contrary, it must be taken into consideration that software markets are subject to special rules and the best solution does not always become the standard. As we discussed in detail in [Chap. 2](#), there are considerable lock-in-effects on software markets. Due to the penguin effect, changing to a different product incurs high risks and switching costs.

What this means for providers of ERP software is that ultimately, having a good product is not enough. It is vitally important to market software solutions in attractive, customer-friendly packages. Economic simulation models reveal that on software markets, it is more effective to increase market share through pricing than by adding more functionality to a product (Buxmann 2002). The smaller a provider's share of the market in comparison to the market leader, the more this holds true. And it begs the question why some providers are offering their open source ERP software at prices comparable to those of SAP.

Judging by the way some providers present themselves, they still have some catching up to do in terms of professional communications and sales strategies. Against this background, it will not be easy for open source ERP software to capture market share from established ERP providers.

And yet, few experts could have predicted that OS software would become as widespread and popular as it is today in various segments of the operating-system market. Time will tell whether the open source community will shake up the software industry again in the ERP space.

References

- Acuña, S., Juristo, N., & Moreno, A. (2006). Emphasizing human capabilities in software development. *IEEE Software*, 23, 94–101.
- Adams, W., & Yellen, J. (1976). Commodity bundling and the burden of monopoly. *The Quarterly Journal of Economics*, 90, 475–498.
- Ahtiala, P. (2006). The optimal pricing of computer software and other products with high switching costs. *International Review of Economics & Finance*, 15, 202–211.
- Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2004). *Web services—concepts, architectures and applications*. Springer, Berlin.
- Altmann, J., Ion, M., Bany, M., & Ashraf, A. (2007). Taxonomy of grid business models. In D. J. Veit & J. Altmann (Eds.), *Grid economics and business models, 4th international workshop, GECON 2007*, Berlin, Heidelberg, pp. 29–43.
- Amberg, M., & Wiener, M. (2006). *IT-offshoring*. Heidelberg: Physika.
- Armstrong, M. (2006). Competition in two-sided markets. *Journal of Economics*, 37, 668–691.
- Arthur, W. B. (1989). Competing technologies, increasing returns, and lock-in by historical events. *The Economic Journal*, 99, 116–131.
- Aspray, W., Mayadas, F., & Vardi, M. (2006). *Globalization and offshoring of software*. New York: Association for Computing Machinery.
- Bakos, Y., & Brynjolfsson, E. (1999). Bundling information goods: Pricing, profits, and efficiency. *Management Science*, 45, 1613–1630.
- Baldwin, C. Y., & Clark, K. B. (1997). Managing in an age of modularity. *Harvard Business Review*, 75, 84–93.
- Baligh, H., & Richartz, L. (1967). *Vertical market structures*. Boston: Allyn and Bacon.
- Balzert, H. (2000). *Lehrbuch der Software-Technik. Software-Entwicklung* (2nd ed.). Heidelberg: Spektrum Akademischer Verlag.
- Bansler, J. P., & Havn, E. C. (2002). Exploring the role of network effects in IT implementation: The case of knowledge management systems. *Proceedings of the 10th European Conference on Information Systems, Information Systems and the Future of the Digital Economy*, Gdansk, Poland, pp 817–829.
- Bauer, S., & Stickel, E. (1998). Auswirkungen der Informationstechnologie auf die Entstehung kooperativer Netzwerkorganisationen. *Wirtschaftsinformatik*, 40, 434–442.
- Beecham, S., Baddoo, N., Hall, T., Robinson, H., & Sharp, H. (2007). Motivation in software engineering: A systematic literature review. *Information and Software Technology*, 50, 860–878.
- Beck, K., & Boehm, B. (2003). Agility through discipline: A debate. *Computer*, 36, 44–46.
- Benlian, A. & Hess, T. (2010). The risks of sourcing software as a service—an empirical analysis of adopters and non-adopters. In: *Proceedings of the 18th European Conference on Information Systems*, Pretoria, South Africa.
- Benlian, A., Hess, T., & Buxmann, P. (2009). Drivers of SaaS-adoption: An empirical study of different application types. *Business & Information Systems Engineering*.
- Benlian, A., Hess, T., & Buxmann, P. (2010). Chancen und Risiken des Einsatzes von SaaS—die Sicht der Anwender. *Wirtschaftsinformatik und Management*, 2, 23–32.

- Berlecon Research. (2002). *Use of open software in firms and public institutions: Evidence from Germany, Sweden and UK*. Berlin: Author.
- Besen, S. M., & Farrell, J. (1994). Choosing how to compete: Strategies and tactics in standardization. *Journal of Economic Perspectives*, 8, 117–131.
- Bhargava, K., & Choudhary, V. (2008). When is versioning optimal for information goods? *Management Science*, 54, 1029–1035.
- Binner, H. F. (1987). *Anforderungsgerechte Datenermittlung für Fertigungssteuerungssysteme*. Cologne: Beuth.
- Blackwell, R. D., Miniard, P. W., & Engel, J. F. (2003). *Consumer behavior*. Orlando, FL: Harcourt.
- Boehm, B. (1986). A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 11, 14–24.
- Boehm, B. (2002). Get ready for agile methods, with care. *Computer*, 35, 1–7.
- Boehm, B. (2006). A view of 20th and 21st century software engineering, pp 12–29.
- Boehm, B., & Turner, R. (2003). *Balancing agility and discipline: A guide for the perplexed*. Boston: Addison-Wesley Longman.
- Boes, A., & Schwemmler, M. (2004). *Herausforderung Offshoring*. Düsseldorf: Hans-Böckler-Stiftung.
- Boes, A., & Schwemmler, M. (2005). *Bangalore statt Böblingen?* Hamburg: VSA.
- Bontis, N., & Chung, H. (2000). The evolution of software pricing: From box licenses to application service provider models. *Electronic Networking Applications and Policy*, 10, 246–255.
- Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., et al. (2004). Web services architecture. W3C working group note. Retrieved from <http://www.w3.org/TR/2004/NOTE-ws-arch-20040211>
- Boysen, N., & Scholl, A. (2009). A general solution framework for component commonality problems. *BuR—Business Research*, 2/1, 86–106.
- Bräutigam, P., & Grabbe, H. (2004). Rechtliche Ausgangspunkte. In P. Bräutigam (Ed.), *IT-outsourcing* (pp. 161–202). Berlin: Erich Schmidt Verlag.
- Brandenburger, A., & Nalebuff, B. (1996). *Co-opetition. A revolutionary mindset that combines competition and cooperation. The game theory strategy that's changing the game of business*. New York: Doubleday.
- Brandt, B. (2010). *Make-or-Buy bei Anwendungssystemen*. Wiesbaden: Gabler.
- Brügge, B., Harhoff, D., Picot, A., Creighton, O., Fiedler, M., & Henkel, J. (2004). *Open-source-software*. Berlin: Springer.
- Burkard, C., Draisbach, T., Widjaja, T., & Buxmann, P. (2010). Ein Framework zur Erhebung von applikationsspezifischen Metadaten in Online-Marktplätzen—Vorstellung und beispielhafte Anwendung im SaaS-Kontext, Darmstädter Arbeitspapier.
- Buxmann, P. (1996). Standardisierung betrieblicher Informationssysteme. Wiesbaden: Gabler.
- Buxmann, P. (2002). Strategien von Standardsoftware-Anbietern: Eine Analyse auf Basis von Netzeffekten. *Zeitschrift für betriebswirtschaftliche Forschung*, 54, 442–457.
- Buxmann, P., Brandt, B., von Ahsen, A., & Hess, T. (2010). Outsourcing der Entwicklung und Anpassung von Anwendungssoftware: Analyse der Kundenzufriedenheit auf Basis einer empirischen Untersuchung, Darmstädter Arbeitspapier.
- Buxmann, P., Diefenbach, H., & Hess, T. (2008a). Die Softwareindustrie. Ökonomische Prinzipien, Strategien, Perspektiven. Wiesbaden: Springer Verlag.
- Buxmann, P., Gerlach, J., & Ristl, J. (2009). Ökonomie von Business-Open-Source-Software am Beispiel ERP. *Linux-Magazin*, 1, 51–54.
- Buxmann, P., Hess, T., & Lehmann, S. (2008b). Software as a service. *Wirtschaftsinformatik*, 50, 500–503.
- Buxmann, P., König, W., Fricke, M., Hollich, F., Martín Díaz, L., & Weber, S. (2004). *Inter-organizational cooperation with SAP solutions—design and management of supply networks* (2nd ed.). Berlin: Springer.

- Buxmann, P., & Lehmann, S. (2009). Preisstrategien von Softwareanbietern. *Wirtschaftsinformatik*, 51, 519–529.
- Buxmann, P., & Matz, J. (2009). ERP-software: Von der Kathedrale zum Basar. *Controlling und Management*, 3, 18–23 (special edition).
- Buxmann, P., & Schade, S. (2007). Wie viel Standardisierung ist optimal? Eine analytische und simulative Untersuchung aus Anwenderperspektive. In U. Blum & A. Eckstein (Eds.), *Wirtschaftsinformatik im Fokus der modernen Wissensökonomik—Festschrift für Prof. Dr. Dr. h.c. Wolfgang Uhr* (pp. 67–88). Dresden: TUDpress.
- Buxmann, P., Strube, J., & Pohl, G. (2007). Cooperative pricing in digital value chains—the case of online music. *Journal of Electronic Commerce Research*, 8, 32–40.
- Buxmann, P., Weitzel, T., & König, W. (1999). Auswirkung alternativer Koordinationsmechanismen auf die Auswahl von Kommunikationsstandards. *Zeitschrift für Betriebswirtschaft, Ergänzungsheft*, 2, 133–151.
- Buxmann, P., Wüstner, E., & Kunze, S. (2005). Wird XML/EDI traditionelles EDI ablösen? Eine Analyse auf Basis von Netzeffekten und einer empirischen Untersuchung. *Wirtschaftsinformatik*, 47, 413–421.
- Campbell-Kelly, M. (1995). Development and structure of the international software industry 1950–1990. *Business and Economic History*, 24, 73–110.
- Capretz, L. F. (2003). Personality types in software engineering. *International Journal of Human Computer Studies*, 58, 207–214.
- Choi, J. (1994). Network externality, compatibility choice, and planned obsolescence. *Journal of Industrial Economics*, 42, 167–182.
- Choudhary, V. (2007). Software as a service: Implications for investment in software development. *Proceedings of the 40th Hawaii International Conference on System Sciences (HICSS 2007)*, Hawaii, 2007.
- Clemons, E. K., Reddi, S. P., & Row, M. C. (1993). The impact of information technology on the organization of economic activity—the “Move to the middle” hypothesis. *Journal of Management Information Systems*, 10, 9–35.
- Condrau, F. (2005). *Die Industrialisierung in Deutschland*. Darmstadt: Wissenschaftliche Buchgesellschaft.
- Costa, P. T., McCrae, R. R., & Holland, J. L. (1984). Personality and vocational interests in an adult sample. *Journal of Applied Psychology*, 69, 390–400.
- Cusumano, M. A. (2004). *The business of software: What every manager, programmer and entrepreneur must know to succeed in good times and bad*. New York: Simon & Schuster Inc.
- Cusumano, M. A. (2007). The changing labyrinth of software pricing. *Communications of the ACM*, 50, 19–22.
- Cusumano, M., & Gawer, A. (2002). The elements of platform leadership. *MIT Sloan Management Review*, 43, 51–58.
- David, P. A. (1985). Clio and the economics of qwerty. *The American Economic Review*, 75, 332–337.
- Delmonte, A. J., & McCarthy, R. V. (2003). Offshore software development: Is the benefit worth the risk? *Proceedings of the 2003 America’s Conference on Information Systems (AMCIS 2003)*, Tampa (Florida), pp 1607–1613.
- Dietrich, O. (1999). In den Tiefen des Kernel. Interview mit Linux-Entwickler Alan Cox. In: c’t 25: 34.
- Diller, H. (2008). *Preispolitik*. Stuttgart: Kohlhammer.
- Dogs, C., & Klimmer, T. (2005). *Agile Software-Entwicklung kompakt*. Bonn: Mitp-Verlag.
- Domschke, W., & Wagner, B. (2005). Models and methods for standardization problems. *European Journal of Operations Research*, 162, 713–726.
- Dostal, W., Jeckle, M., Melzer, I., & Zengler, B. (2005). Service-orientierte Architekturen mit Web Services—Konzepte, Standards, Praxis. Heidelberg u. a.: Spektrum.
- Earl, M. J. (1996). The risks of outsourcing IT. *Sloan Management Review*, 37, 26–32.

- Eisenmann, T. (2008). Managing proprietary and shared platforms. *California Management Review*, 50, 31–53.
- Eisenmann, T., Parker, G., & Van Alstyne, M. (2009). Opening platforms: How, when and why? In: Gawer (Ed.). *Platforms, markets and innovation*. London: Edward Elgar.
- Eisenmann, T., & Wong, J. (2004). Electronic arts in online gaming. Harvard Business School Case 804-140.
- Erl, T. (2006). *Service-oriented architecture—concepts, technology and design*. Prentice hall, Upper Saddle River.
- Farrell, J., & Saloner, G. (1985). Standardization, compatibility, and innovation. *Rand Journal of Economics*, 16, 70–78.
- Farrell, J., & Saloner, G. (1987). Competition, compatibility, and standards: The economics of horses, penguins and lemmings. In H. L. Gabel (Ed.), *Product standardization and competitive strategy* (pp. 1–22). Amsterdam: North Holland.
- Fitzgerald, B., Hartnett, G., & Conboy, K. (2006). Customising agile methods to software practices at Intel Shannon. *European Journal of Information Systems*, 15, 200–213.
- Florissen, A. (2008). Preiscontrolling—Rationalitätssicherung im Preismanagement. *Zeitschrift für Controlling und Management*, 52, 85–90.
- Fowler, M., & Highsmith, J. (2001). The Agile Manifesto. *Software Development*, 9(8), 28–32.
- Franck, E. (2003). Open Source aus ökonomischer Sicht. *Wirtschaftsinformatik*, 45, 527–532.
- Friedewald, M., Rombach, H. D., Stahl, P., Broy, M., Hartkopf, S., Kimpeler, S., Kohler, K., Wucher, R., & Zoche, P. (2001). Analyse und Evaluation der Softwareentwicklung in Deutschland. Studie für das Bundesministerium für Bildung und Forschung (BMBF). Nuremberg: GfK Marktforschung.
- Friedman, T. (2005). *The world is flat: A brief history of the 21st century*. London: Penguin Books.
- Gadatsch, A. (2006). *IT-Offshore realisieren*. Wiesbaden: Vieweg.
- Gawer, A. (2009). Platform dynamics and strategies: From products to services. In: Gawer, A. (Ed.). *Platforms, markets and innovation* (pp. 45–76).
- Gawer, A., & Cusumano, M. A. (2002). *Platform leadership: How Intel, Microsoft, and Cisco Drive Industry Innovation*. Boston, MA: Harvard Business School Press.
- Government of India, Ministry of Communications & Information Technology (Ed.). (2006). *Information technology: Annual report 2005–06*, New Delhi. Retrieved from <http://www.mit.gov.in/annualreport2005-06.pdf>
- Grassmuck, V. (2004). *Freie Software—Zwischen Privat- und Gemeineigentum*. Bonn: Bundeszentrale für politische Bildung.
- Gregory, R. W. (2010). Review of th0e IT offshoring literature: The role of cross-cultural differences and management practices. *18th European Conference on Information Systems*, Manuscript ID: ECIS2010-0086.R1.
- Günther, O., Tamm, G., Hansen, L., & Meseg, T. (2001). Application service providers: Angebot, Nachfrage und langfristige Perspektiven. *Wirtschaftsinformatik*, 43, 555–568.
- Harmon, R., Raffo, D., & Faulk, S. (2005). Value-based pricing for new software products: Strategy insights for developers. Retrieved from <http://www.cpd.ogi.edu/MST/CapstoneSPR2005/VBSP.pdf>
- Hecker, F. (1999). Setting up shop: The business of open-source-software. *IEEE Software*, 16, 45–51.
- Heinen, E. (1991). *Industriebetriebslehre—Entscheidungen im Industriebetrieb*. Wiesbaden: Gabler.
- Heinrich, B., Klier, M., & Bewernik, M. (2006). Unternehmensweite Anwendungsintegration—Zentrale Anreizsetzung zur Realisierung von Netzeffekten bei dezentralen Entscheidungsstrukturen. *Wirtschaftsinformatik*, 48, 158–168.
- Henkel, J. (2004): The Jukebox Mode of innovation—a model of commercial open source development. CEPR discussion paper 4507.

- Henn, J., & Khan, A. (2007). Serviceorientierung—Mehr als nur IT-Architekturen. In: Fink, D., Gries, A., & Lünendonk, T. (Ed.). *Consulting-Kompodium* (pp. 208–217). FAZ Frankfurt.
- Hess, T., Buxmann, P., Mann, F., & Königer, M. (2007). Industrialisierung der Softwarebranche: Erfahrungen deutscher Anbieter. *Management Reports des Instituts für Wirtschaftsinformatik und Neue Medien 2*, Munich.
- Hess, T., & Schumann, M. (1999). *Medienunternehmen im digitalen Zeitalter—Neue Technologien—Neue Märkte—Neue Geschäftsansätze*. Wiesbaden: Gabler.
- Hess, T., & Ünlü, V. (2004). Systeme für das Management digitaler Rechte. *Wirtschaftsinformatik*, 46, 273–280.
- Hill, S. (2008). SaaS economics seem to favor users more than vendors. *Manufacturing Business Technology*, p. 48.
- Hindel, B., Hörmann, K., Müller, M., & Schmied, J. (2004). *Basiswissen Software-Projektmanagement*. Heidelberg: Dpunkt Verlag GmbH.
- Hirnlé, C., & Hess, T. (2007). Investing into IT infrastructures for inter-firm networks: Star Alliance's move to the common platform. *Electronic Journal for Virtual Organizations and Networks*, 8, 124–143.
- Hirschheim, R., Heinzl, A., & Dibbern, J. (2006). *Information systems outsourcing*. Heidelberg: Springer.
- Hitt, L. M., & Chen, P. (2005). Bundling with customer self-selection: A simple approach to bundling low marginal cost goods. *Management Science*, 51, 1481–1493.
- Hoch, D. J., Roeding, C. R., Purkert, G., & Lindner, S. K. (2000). *Erfolgreiche Software-Unternehmen. Die Spielregeln der New Economy*. Munich: Hanser.
- Homburg, C., & Koschate, N. (2005). Behavioral Pricing—Forschung im Überblick. *Zeitschrift für Betriebswirtschaft*, 75, 383–423 and 501–524.
- Homburg, C., & Krohmer, H. (2006). *Marketing management*. Wiesbaden: Gabler.
- Howcroft, D., & Light, B. (2006). Reflections on issues of power in packaged software selection. *Information Systems Journal*, 16, 215–235.
- Hutzschenreuter, T., & Stratigakis, G. (2003). Die feindliche Übernahme von PeopleSoft durch Oracle—der Beginn einer Konsolidierungswelle. *Signale*, 18(3), 10–11.
- Ichbiah, D. (1993). *Die Microsoft Story: Bill Gates und das erfolgreichste Softwareunternehmen der Welt*. Munich: Campus.
- Izci, E., & Schiereck, D. (2010). Programmierte Wertgenerierung durch M&A in der Business-Softwareindustrie? *Mergers and Acquisitions Review*, 2, 69–74.
- Jadhav, A. S., & Sonar, R. M. (2009). Evaluating and selecting software packages: A review. *Information & Software Technology*, 51, 555–563.
- Jensen, M. C., & Meckling, W. H. (1976). Theory of the firm: Managerial behaviour, agency costs and ownership structure. *Journal of Financial Economics*, 3, 305–360.
- Katz, M. L., & Shapiro, C. (1985). Network externalities, competition, and compatibility. *American Economic Review*, 75, 424–440.
- Kearney, A. T. (2004). *Making offshore decisions*. Chicago: Offshore Location Attractiveness Index.
- Keil, M., & Tiwana, A. (2006). Relative importance of evaluation criteria for enterprise systems: A conjoint study. *Information Systems Journal*, 16, 237–262.
- Kern, T., Willcocks, L. P., & Lacity, M. C. (2002). Application service provision: Risk assessment and mitigation. *MIS Quarterly Executive*, 1, 113–126.
- Kittlaus, H.-B., & Clough, P. N. (2009). *Software product management and pricing. Key success factors for software organizations*. Berlin: Springer.
- Kittlaus, H.-B., Rau, C., & Schulz, J. (2004). *Software-Produkt-Management: Nachhaltiger Erfolgsfaktor bei Herstellern und Anwendern*. Heidelberg: Springer.
- Köhler, L. (2004). *Produktinnovation in der Medienindustrie - Organisationskonzepte auf Basis von Produktplattformen*. Wiesbaden: Gabler.

- Kollock, P. (1999). The economies of online cooperation: Gifts and public goods in cyberspace. In M. A. Smith & P. Kollock (Eds.), *Communities in cyberspace* (pp. 220–239). London: Routledge.
- Krafzig, D., Banke, K., & Slama, D. (2006). *Enterprise SOA. Best Practices für Serviceorientierte Architekturen—Einführung, Umsetzung, Praxis*. Upper Saddle River u. a: Prentice Hall PTR.
- Kublanov, E. M., Satyaprasad, S., & Nambiyattil, R. (2005). Offshore & Nearshore ITO salary report 2004, Vol. 3, neoIT, San Ramon. Retrieved from http://www.neoit.com/pdfs/whitepapers/Olv3i05_0505_ITO-Salaries2004.pdf
- Küpper, H.-U. (2008). *Controlling. Konzeption, Aufgaben, Instrumente*. Stuttgart: Schäffer-Poeschel.
- Lambrecht, A., & Skiera, B. (2006). Paying too much and being happy about it: Existence, causes, and consequences of tariff-choice biases. *Journal of Marketing Research*, 43, 212–223.
- Larman, C., & Basili, V. R. (2003). Iterative and incremental development: A brief history. *Computer*, 36, 47–56.
- Lauver, K. J., & Kristof-Brown, A. (2001). Distinguishing between employees' perceptions of person–job and person–organization fit. *Journal of Vocational Behavior*, 59, 454–470.
- Lee, S. Y. T., Kim, H.-W., & Gupta, S. (2009). Measuring open source software success. *Omega—The International Journal of Management Science*, 37, 426–438.
- Lehmann, S., & Buxmann, P. (2009). Preisstrategien von Softwareanbietern. *Wirtschaftsinformatik*, 51, 519–529.
- Lehmann, S., Draibach, T., Koll, C., Buxmann, P., & Diefenbach, H. (2010). Preisgestaltung für Software-as-a-Service. Ergebnisse einer empirischen Analyse mit Fokus auf nutzungsabhängige Preismodelle. *Proceedings zur Teilkonferenz "Software-Industrie" der Multikonferenz Wirtschaftsinformatik (MKWI) 2010*, pp. 505–516.
- Leimbach, T. (2007). Vom Programmierbüro zum globalen Softwareproduzenten. Die Erfolgsfaktoren der SAP von der Gründung bis zum R/3-Boom. *Zeitschrift für Unternehmensgeschichte*, 52, 5–34.
- Lerner, J., & Tirole, J. (2002). Some simple economics of open source. *Journal of Industrial Economics*, 52, 197–234.
- Liebowitz, S. J., & Margolis, S. E. (1994). Network externality: An uncommon tragedy. *Journal of Economic Perspectives*, 8, 133–150.
- Liebowitz, S. J., & Margolis, S. E. (2001). *Winners, losers & Microsoft competition and antitrust in high technology*. Oakland: Independent Institute.
- Linde, F. (2008). *Pricing-Strategien bei Informationsgütern*. *WISU*, 2, 208–214.
- Lock, D. (1998). *The Gower handbook of management*. Hampshire: Gower Publishing.
- Loefert, C. (2007). *Unternehmensreputation und M&A-Transaktionen—Bewertung strategischer Entscheidungen in der US-amerikanischen Finanz- und Telekommunikationsindustrie*. Wiesbaden: Gabler.
- Lünendonk, T. (2007). *Führende Standard-Software-Unternehmen in Deutschland*. Bad Wörishofen.
- Lünendonk, T. (2009). *Führende IT-Beratungs-, IT-Service- und Standard-Software-Unternehmen in Deutschland*. Bad Wörishofen.
- Marn, M. V., Roegner, E. V., & Zawada, C. C. (2003). *The power of pricing*. The McKinsey Quarterly, Number 1.
- Martin Diaz, L. (2006). *Evaluation of cooperative planning in supply chains. An empirical approach of the European Automotive Industry*. Wiesbaden: Gabler.
- Mathwick, C., Malhotra, N., & Rigdon, E. (2001). Experiential value: Conceptualization, measurement and application in the catalog and internet shopping environment. *Journal of Retailing*, 77, 39–56.
- Mell, P., Grance, T. (2011). The NIST-definition of cloud computing. Information note dated 25th May 2012. Retrieved from http://www.nist.gov/manuscript-publication-search.cfm?pub_id=909616

- Mergerstat Free Reports. (2009). Mergerstats free reports: Industry rankings for year 2009. Retrieved from https://www.mergerstat.com/newsite/free_report.asp
- Mertens, P., Große-Wilde, J., & Wilkens, I. (2005). *Die (Aus-)Wanderung der Softwareproduktion—Eine Zwischenbilanz* (Vol. 38). Erlangen-Nuremberg: Institut für Informatik der Friedrich-Alexander-Universität Erlangen-Nuremberg.
- Messerschmitt, D. G., & Szyperki, C. (2003). *Software ecosystem. Understanding an indispensable technology and industry*. Cambridge: MIT Press.
- Meyer, M. H., & Lehnerd, A. P. (1997). *The power of product platforms: Building value and cost leadership*. New York: Free Press.
- Meyer, R. (2008). *Partnering with SAP Vol. 1: Business models for software companies*. Norderstedt: Books on Demand.
- Meyer-Stamer, J. (1999). *Lokale und regionale Standortpolitik—Konzepte und Instrumente jenseits von Industriepolitik und traditioneller Wirtschaftsförderung*. Institut für Entwicklung und Frieden, Gerhard-Mercator-Universität Duisburg.
- Miller, D., & Elgård, P. (1998). Defining modules, modularity and modularization. *Proceedings of the 13th IPS Research Seminar*, Fuglsoe.
- Mit Siebel kauft Oracle Marktanteile. (2005). *Computerwoche*. Retrieved from <http://www.computerwoche.de/nachrichten/566287>
- Morstead, S., & Blount, G. (2003). *Offshore ready: Strategies to plan and profit from offshore IT-enabled services*. Houston: ISANI Press.
- Mundhenke, J. (2007). *Wettbewerbswirkungen von Open-Source-Software und offenen Standards auf Softwaremärkten*. Springer, Berlin.
- Nalebuff, B. (2004). *Bundling as an entry barrier*. Working paper. School of Management, Yale University, New Haven.
- NASSCOM. (2007). NASSCOM's education initiatives. Sustaining India's talent edge to fuel the next wave of IT-BPO industry growth. NASSCOM Press. Information note dated July 5, 2007. Retrieved from <http://www.nasscom.in/upload/5216/July%205%202007%20%20Education%20/initiatives-Final.pdf>
- Ney, J. (2004). *Power in the global information age*. London: Routledge.
- Nieschlag, R., Dichtl, E., & Hörschgen, H. (2002). *Marketing* (19th ed.). Berlin: Duncker & Humblot.
- Olderog, T., & Skiera, B. (2000). The benefits of bundling strategies. *Schmalenbach Business Review*, 1, 137–160.
- Ouchi, W. G. (1977). The relationship between organizational structure and organizational control. *Administrative Science Quarterly*, 3, 95–113.
- Osterwalder, A. (2004). *The business model ontology—a proposition in a design science approach*. Dissertation, University of Lausanne.
- Parbel, M. (2005). Symantec übernimmt Veritas: Security und Storage wachsen zusammen. Retrieved from <http://www.crn.de/showArticle.jhtml?articleID=184423707>
- Payscale. (2010). Salary benchmarking. Retrieved from www.payscale.com
- Pepels, W. (1998). *Einführung in das Preismanagement*. Munich: Oldenbourg.
- Perens, B. (1999). The open source definition. In C. DiBona, S. Ockman, & M. Stone (Eds.), *Open sources: Voices from the open source revolution* (pp. 171–188). Sebastopol: O'Reilly.
- Petrasch, R., & Meimberg, O. (2006). *Model driven architectures*. Heidelberg: dpunkt.
- Picot, A. (1982). Transaktionskostenansatz in der Betriebswirtschaftslehre: Stand der Diskussion und Aussagewert. *Die Betriebswirtschaft*, 42, 267–284.
- Picot, A. (1989). Zur Bedeutung allgemeiner Theorieansätze für die betriebswirtschaftliche Information und Kommunikation. In W. Kirsch & A. Picot (Eds.), *Die Betriebswirtschaftslehre im Spannungsfeld zwischen Generalisierung und Spezialisierung* (pp. 361–379). Wiesbaden: Gabler.
- Picot, A., & Ertsey, B. (2004). IT-Management der Zukunft als Vertragsmanagement. *Information Management & Consulting*, 19, 11–19.

- Picot, A., & Meier, M. (1992). Analyse- und Gestaltungskonzepte für das Outsourcing. *Information Management*, 7(4), 14–27.
- Pigou, A. C. (1929). *The economics of welfare* (3rd ed.). London: Macmillan.
- Plattner, H. (2007). Trends and concepts, lecture. Retrieved from http://epic.hpi.uni-potsdam.de/Home/TrendsAndConcepts_I_2007
- Prehl, S. (2006). Der Offshore-Trend erreicht Europa. *Computerwoche Online*, 18, 38.
- Raymond, E. S. (1999). *The Cathedral and the Bazaar*. Sebastopol: O'Reilly.
- Reussner, R., & Hasselbring, W. (2006). Handbuch der Software-Architektur. Heidelberg: dpunkt.
- Robinson, M., & Kalakota, R. (2005). *Offshore outsourcing*. Alpharetta: Mivar Press.
- Ross, S. A. (1973). The economic theory of agency: The principal's problem. *The American Economic Review*, 63, 134–139.
- Schade, S., & Buxmann, P. (2005). A prototype to analyse and support standardization decisions. In T. M. Egyedi & M. H. Sherif (Eds.), *Proceedings of the 4th International Conference on Standardization and Innovation in Information Technology*, September 21–23 2005 (pp. 207–219). Geneva: ITU.
- Schmalensee, R. (1984). Gaussian demand and commodity bundling. *The Journal of Business*, 57, 211–230.
- Schmidt, C., Weinhardt, C., & Horstmann, R. (1998). Internet-Auktionen—Eine Übersicht für Online-Versteigerungen im Hard- und Softwarebereich. *Wirtschaftsinformatik*, 40, 450–457.
- Schmidt, D. C. (2006). Model-driven-engineering. *IEEE Computing*, 39, 25–31.
- Schweitzer, M. (1994). *Industriebetriebslehre—Das Wirtschaften in Industrieunternehmen*. Munich: Vahlen.
- SEI. (2010). People CMM. Retrieved from <http://www.sei.cmu.edu/cmmt/tools/peoplecmm>
- Shapiro, C., & Varian, H. R. (1998). *Information rules: A strategic guide to the network economy*. Boston: Harvard Business School Press.
- Shapiro, C., & Varian, H. R. (1999). *Information rules: A strategic guide to the network economy*. Boston: Harvard Business School.
- Sharma, S., Sugumaran, V., & Rajagopalan, B. (2002). A framework for creating hybrid-OSS communities. *Information Systems Journal*, 12, 7–25.
- Sieg, G. (2005). *Spieltheorie*. Munich: Oldenbourg.
- SIIA, MACROVISION, SOFTSUMMIT, SVPMA, CELLUG. (2006). Key trends in software pricing and licensing: A survey of software industry executives and their enterprise customers. Retrieved from http://softsummit.com/pdfs_registered/SW_Pricing_Licensing_Report_2006_2007.pdf
- Simon, H. (1992). *Preismanagement*. Stuttgart: Gabler.
- Simonson, I., & Tversky, A. (1992). Choice in context: Tradeoff contrast and extremeness aversion. *Journal of Marketing Research*, 29, 281–295.
- Skiera, B. (1999a). Preisdifferenzierung. In S. Albers, M. Clement, & K. Peters (Eds.), *Marketing mit interaktiven Medien. Strategien zum Markterfolg* (pp. 283–296). Frankfurt am Main: F.A.Z. Institut.
- Skiera, B. (1999b). *Mengenbezogene Preisdifferenzierung bei Dienstleistungen*. Wiesbaden: Gabler.
- Skiera, B. (2000). Wie teuer sollen die Produkte sein?—Preispolitik, Ecommerce: Einstieg, Strategie und Umsetzung im Unternehmen. In: S. Albers, M. Clement, K. Peters & B. Skiera (Eds.) eCommerce. Einstieg, Strategie und Umsetzung in Unternehmen. Frankfurt: Frankfurter Allgemeine Buch, S 95–108.
- Skiera, B., & Spann, M. (1998). Gewinnmaximale zeitliche Preisdifferenzierung für Dienstleistungen. *Zeitschrift für Betriebswirtschaft*, 68, 703–718.
- Skiera, B., & Spann, M. (2000). Flexible Preisgestaltung im Electronic Business. In R. Weiber (Ed.), *Handbuch electronic business: Informationstechnologien—electronic commerce—Geschäftsprozesse* (pp. 539–557). Wiesbaden: Gabler.

- Skiera, B., & Spann, M. (2002). Preisdifferenzierung im Internet. In M. Schögel, T. Tomczak, & C. Belz (Eds.), *Roadmap to E-Business—Wie Unternehmen das Internet erfolgreich nutzen* (pp. 270–284). St. Gallen: Thexis.
- Skiera, B., Spann, M., & Walz, U. (2005). Erlösquellen und Preismodelle für den Business-to-Consumer-Bereich im Internet. *Wirtschaftsinformatik*, 47, 285–293.
- Smith, G. E., & Nagle, T. T. (1995). Frames of reference and buyers' perception of price and value. *California Management Review*, 38, 98–116.
- Spence, A. M., & Zeckhauser, R. J. (1971). Insurance, information and individual action. *American Economic Review*, 61, 380–391.
- Sterne, P., & Herring, N. (2006). SugarCRM—a sweet mix of commercial and open source. Linux. Retrieved 16 Nov 2006 from <http://linux.sys-con.com/node/173436>
- Stremersch, S., & Tellis, G. J. (2002). Strategic bundling of products and prices: A new synthesis for marketing. *Journal of Marketing*, 66, 55–72.
- Suermann, J. C. (2006). Bilanzierung von Software nach HGB, US-GAAP und IFRS—Integrative Analyse der Regelungen zu Ansatz, Bewertung und Umsatzrealisation von Software aus Hersteller- und Anwendersicht. Diss. Universität Würzburg. Retrieved from <http://www.opus-bayern.de/uni-wuerzburg/volltexte/2006/1933>
- Sundararajan, A. (2004). Nonlinear pricing of information goods. *Management Science*, 50, 1660–1673.
- “Temporary Workers”. U.S. Department of State. (2006). http://travel.state.gov/visa/temp/types/types_1271.html
- The Standish Group International, Inc. (2009). New Standish Group report shows more project failing and less successful projects. Retrieved from http://www1.standishgroup.com/newsroom/chaos_2009.php. Pressemeldung vom 23.04.2009, Massachusetts.
- Thondavadi, N., & Albert, G. (2004). *Offshore outsourcing*. Bloomington: Istbooks.
- Unilog Integrierte Unternehmensberatung GmbH. (2002). *Client Studie der Landeshauptstadt München, Munich*.
- Valtakoski, A., & Rönkkö, M. (2009). Business models of software firm. *Proceedings of the HICSS*.
- Van Antwerp, M., & Madey, G. (2008). Advances in the SourceForge Research Data Archive (SRDA). *Fourth International Conference on Open Source Systems, IFIP 2.13 (WoPDaSD 2008)*
- Van der Linden, F., Bosch, J., Kamsties, E., Käsälä, K., & Obbink, H. (2004). Software product family evaluation. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, et al. (Eds.), *Lecture notes in computer science* (pp. 110–129). Berlin: Springer.
- Van Westendorp, P. H. (1976). NSS-price sensitivity meter: A new approach to study consumer perception of prices. *Venice ESOMAR Congress* (pp. 139–167), Amsterdam.
- Vaquero, L. M., Rodero-Merino, L., Caceres, J., & Lindner, M. (2009). A break in the clouds: Towards a cloud definition. *SIGCOMM Computer Communication Review ACM*, 39, 50–55.
- Varian, H. R. (1997). Versioning information goods. Working paper, University of California, Berkeley. Retrieved from <http://people.ischool.berkeley.edu/~hal/Papers/version.pdf>
- Vashistha, A., & Vashistha, A. (2006). *The offshore nation*. New York: McGraw-Hill.
- Viswanathan, S., & Anandalingam, G. (2005). Pricing strategies for information goods. *Sadhana—Journal of the Indian Academy of Sciences*, 30, 257–274.
- Weber, J., & Schäffer, U. (2008). *Einführung in das Controlling*. Stuttgart: Schäffer-Poeschel.
- Weitzel, T. (2004). *Economics of standards in information networks*. Heidelberg: Physica.
- Weitzel, T., Wendt, O., & von Westarp, F. (2000). Reconsidering network effect theory. *Proceedings of the 8th European conference on information systems (ECIS 2000)*, Vienna, pp. 484–491.
- West, J. (2003). How open is open enough? Melding proprietary and open source platform strategies. *Research Policy*, 32, 1259–1285.

- West, J. (2006). The economic realities of open standards: Black, white and many shades of gray. In S. Greenstein & V. Stango (Eds.), *Standards and public policy*. Cambridge: Cambridge University Press.
- Wheelwright, S. C., & Clark, K. B. (1992). Creating project plans to focus product development. *Harvard Business Review*, 70, 67–83.
- Willcocks, L., & Lacity, M. C. (2006). *Global sourcing of business and IT services*. New York: Palgrave Macmillan.
- Williamson, O. E. (1985). The economic institutions of capitalism. Firms, markets, relational contracting. New York: Free Press.
- Williamson, O. E. (1991). Comparative economic organization: The analysis of discrete structural alternatives. *Administrative Science Quarterly*, 36, 269–296.
- Wirtz, B. W. (2003). *Mergers & acquisitions management. Strategie und Organisation von Unternehmenszusammenschlüssen*. Wiesbaden: Gabler.
- Wolf, C. (2010). *Leistungstiefe von Softwareunternehmen—Eine Untersuchung von Herstellern monolithischer Standardanwendungssoftware*. Hamburg: Kovač.
- Wolf, C. M., Benlian, A., & Hess, T. (2010). Industrialisierung von Softwareunternehmen durch Arbeitsteilung: Einzelfall oder Trend? Multikonferenz Wirtschaftsinformatik 2010.
- Wu, S., & Anandalingam, G. (2002). Optimal customized bundle pricing for information goods. *Proceedings workshop on information technology and systems*, Barcelona.
- Wu, S., Hitt, L. M., Chen, P., & Anandalingam, G. (2008). Customized bundle pricing for information goods: A nonlinear mixed-integer programming approach. *Management Science*, 54, 608–622.
- Wüstner, E. (2005). *Standardisierung und Konvertierung: Ökonomische Bewertung und Anwendung am Beispiel von XML/EDI*. Aachen: Shaker.
- Yoffie, D. B., & Kwak, M. (2006). With friends like these: The art of managing complementors. *Harvard Business Review*.
- Zacharias, R. (2007). Produktlinien: Der nächste Schritt in Richtung Software-Industrialisierung. *Java Magazin*, 3, 69–82.
- Zerdick, A., Picot, A., Schrape, K., Artopé, A., Goldhammer, K., Lange, U. T., et al. (1999). *Die Internet-Ökonomie: Strategien für die digitale Wirtschaft*. Berlin: Springer.
- Zhang, Q., Cheng, L., & Boutaba, R. (2010). Cloud computing: State-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(2), 7–18.

Index

A

Accenture, 114, 121–122
Adobe, 88, 94–95
Application engineering, 159
Application erontend, 148–149
Application service providing (ASP), 169, 176
Automation, 160–161

B

Backsourcing, 129, 131
Best-of-breed, 37, 41
Billing units, 84–85, 96–97, 186–187, 189–190
Broker, 61–63, 172
Browser War, 26–27, 165
Build-operate-transfer (BOT) model, 116
Bundling, 26–27, 81, 88–93, 98
Business model, 3, 9, 14–17, 165, 170, 173–174, 183–185, 187, 189, 200, 202–205
Business relationship, 32, 125, 128, 130

C

Cash flow, 148–150, 152–153, 176–177
Cloud computing, 9, 169–173
CMMI, 107
Communication costs, 35, 41
Compatibility, 10–11, 14, 28, 43
reverse, 163
Complementor, 5, 162–163, 165–166
Computer aided software engineering, 104
Concurrent user, 85, 96, 189
Conjoint analysis, 188
Cooperation strategy, 55–57
Co-opetition, 59–60
Console wars, 30
Consulting, 8, 16–17, 19, 21, 59, 67–68, 72–74, 81, 86, 88, 201
Consulting service, 21, 86

Copyleft, 192

CRM software, 68, 174, 190
Customer relationship management, 140, 202
Customizing, 8, 94, 127–128, 132

D

Digital games industry, 29–30
Digital good, 13, 17, 19–20, 81, 83, 86, 92
Distributor, 47, 65, 75
Distribution channel, 30, 74–75, 162, 165
Distribution system, 75
Domain engineering, 159
Dual licensing model, 202

E

Early adopter, 26
Enterprise resource planning, 96, 98, 140–141, 143, 153, 173, 175, 178, 181–182, 190, 202–210
External service provider, 124

F

Farshoring, 116–117, 120–122, 134–135
First copy, 19, 81
Follow-the-free strategy, 93–94, 200
Follow-the-sun-principle, 117, 120, 138
Free software, 17, 191–193, 195
Free software foundation, 192
Freeware, 191
Function point method, 99–100

G

Global delivery model, 122
GNU, 192–193, 195–196
Google, 12, 62, 155, 164, 170–176
GPL-license, 192–193, 195, 198, 201–202, 204–205

H

Hidden Aation, 49
 Hidden characteristics, 48–49
 Hidden intention, 49

I

IBM, 4, 16, 28–30, 150, 153, 165, 170, 176, 201
 iBS banking solution, 58–59
 Increasing returns, 21, 25
 Industrialization, 47, 106, 159–161
 Information costs, 34–41
 Incompatibility, 69
 Intermediation theory, 47

J

Joint venture, 58, 115–117, 138

K

Key account manager, 76
 Key performance indicator (KPI), 71, 77–80, 85, 96, 148

L

Lessor, 60, 62–63
 LGPL-license, 193, 195, 205
 Linux, 173, 192–193, 196–197, 199–205
 License, 14–17, 19, 30, 60, 72–76, 81, 83–84, 86, 88–89, 94, 98, 122, 164, 169, 174, 177–178, 191–195, 198, 201–202, 204–207
 Local attractiveness index, 121
 Location selection, 114, 121
 Lock-in-effect, 23, 28, 87–88, 93–94, 172–173, 176, 178, 210

M

Maintainer, 197
 Make-or-buy, 41, 45, 115, 144
 Mergers and acquisitions, 3, 64–70
 Microsoft, 4, 6, 22–23, 26–27, 29–30, 34, 60, 69, 87, 90, 142, 153, 155, 163, 165, 170–171, 176, 195, 198–199, 201
 Model-Driven Architecture, 105
 Model-Driven Engineering (MDE), 105–107
 Move to the market, 46
 Move-to-the-middle, 45–46

N

Nash equilibrium, 39–40
 Nearshoring, 116–117, 120–122, 134
 Network Effect, 3, 19–33, 41, 55, 67, 69, 83, 88, 93–94, 164–165, 176, 200
 direct, 22, 24
 indirect, 21–22, 24, 30, 165, 200
 two-sided, 21, 29–31
 Network effect factor, 22–23, 28, 94

O

Openness, 128, 162–165
 horizontal, 163
 vertical, 162–163
 Offshore, 42, 113, 115–122, 134–138, 153
 Offshoring, 45, 113–122, 135, 137
 On-demand, 170, 172, 174–176, 184
 Onshoring, 114, 122
 Open source initiative (OSI), 24, 193, 195, 207
 Open source software (OSS), 22, 56, 191, 193, 195–196, 199–202
 Oracle, 14, 17, 28, 65, 68–69, 153, 174, 201–203
 Outsourcing, 8–9, 43, 45–46, 48–49, 98, 113–139, 172–173, 176–177, 179, 183

P

Partnership, 28, 57–64, 75–76, 115, 121, 126, 144, 164, 201
 development partnership, 58–60
 OEM partnership, 58, 62–63
 referral partnership, 58, 63
 reseller partnership, 58, 60–61
 shared revenue partnership, 58, 61–62
 standardization partnership, 58, 64
 Path dependence, 24–25
 Penguin effect, 20, 23–24, 28, 40, 164, 210
 Person-job-fit, 107–108
 Pirate copy, 20
 Platform sponsor, 164
 Point of marginal cheapness (PMC), 189
 Point of marginal expensiveness (PME), 189
 Positive feedback, 21–22, 26
 Price bundling, 26–27, 81, 88–93, 98
 Price differentiation, 98
 Price sensitivity meter (PSM), 188–189
 Pricing, 17, 23, 50, 55, 58, 74, 81–100, 170, 173, 183–190, 204, 210
 Principal-agent-theory, 47–49, 53
 Product line management, 159
 Product platform, 155–159

R

Return

- abnormal, 70

Reservation price, 86, 91, 189

Rule of thumb, 33, 87

S

SaaS, 59, 74, 83–85, 98, 169–190

Sales

- direct, 72–73, 142, 146

- indirect, 72–73, 142, 145

Salesforce.com, 155, 171–175

Sales management, 71

Sales organization, 71–72

Sales strategy, 71–81

SAP, 4–6, 8, 12, 14, 16, 22, 28, 30, 58–62, 68, 73–76, 113, 119, 121, 123, 150, 153, 173–175, 201–202, 210

Service bus, 149

Service contract, 66, 149

Service repository, 149

Service-oriented architecture (SOA), 6, 33, 41, 59, 138, 146, 148–153, 156, 169

Skimming strategy, 93, 95

Small and mid-sized businesses, 188

SOAP, 150–153

Software

- Custom software, 5, 7–8, 16, 19, 42, 47, 50, 53, 72, 98–99, 104–105, 123–128, 132

- Standard software, 5–10, 14–17, 19, 21–22, 26, 28, 34–35, 47, 58–59, 72–73, 86, 98, 104, 126–128, 133, 140, 146, 153, 169, 173, 178, 199

- System software, 5, 14

Software AG, 65, 115–117, 138, 150, 153

Software as a service, 169–190

Software provider

- in the narrower sense, 4–5, 8–9, 191

- in the broader sense, 4, 8–9, 16–17

Software development

- ad-hoc development, 101

- agile approach, 101–104, 196

- plan-based approach, 101–104

- rapid prototyping, 101

Software license, 19, 75–76, 83–84, 86, 88, 174, 192–193, 195

Standardization problem, 33–41

Stand-alone utility, 22–26, 35–36, 38–40

Startup problem, 24, 93

SuSe, 200–201

Switching costs, 21, 23, 28, 98, 173, 178, 210

T

Time difference, 137–144

Transaction, 19, 41–47, 65, 67–72, 84, 89, 99, 139, 142, 190

Transaction cost, 19, 42–47, 65, 72–73

Transaction cost theory, 41–47, 115, 139

U

UDDI, 150–153

V

Value chain, 14–15, 21, 29–30, 41, 46, 55, 65, 71, 139–146, 153

Value net, 56

W

W3C, 150

Web service, 33, 84, 90, 150–153, 169, 171

Willingness-to-pay, 86–88, 188–189

Winner-takes-it-all-market, 20

WSDL, 150–153

X

XML, 21, 105, 148, 150–151