von Engelhardt, Sebastian

Working Paper

The economic properties of software

Jena Economic Research Papers, No. 2008,045

Provided in Cooperation with:
Max Planck Institute of Economics

Suggested Citation: von Engelhardt, Sebastian (2008) : The economic properties of software, Jena Economic Research Papers, No. 2008,045, Friedrich Schiller University Jena and Max Planck Institute of Economics, Jena

This Version is available at:
https://hdl.handle.net/10419/25729

JENA ECONOMIC RESEARCH PAPERS

2008 – 045

The Economic Properties of Software

by

Sebastian von Engelhardt

www.jenecon.de

ISSN 1864-7057

The JENA ECONOMIC RESEARCH PAPERS is a joint publication of the Friedrich Schiller University and the Max Planck Institute of Economics, Jena, Germany. For editorial correspondence please contact m.pasche@wiwi.uni-jena.de.

Impressum:

Friedrich Schiller University Jena
Carl-Zeiss-Str. 3
D-07743 Jena
www.uni-jena.de

Max Planck Institute of Economics
Kahlaische Str. 10
D-07745 Jena
www.econ.mpg.de

Jena Economic Research Papers 2008-045

The Economic Properties of Software

Sebastian von Engelhardt*
Department of Economics
Carl-Zeiß-Str. 3
D-07743 Jena, Germany

Abstract

Software is a good with very special economic characteristics. Taking a general definition of software as its starting-point, this article systematically elaborates the central qualities of the commodity which have implications for its production and cost structure, the demand, the contestability of software-markets, and the allocative efficiency.

In this context it appears to be reasonable to subsume the various characteristics under the following generic terms: software as a means of data-processing, software as a system of commands or instructions, software as a recombinant system, software as a good which can only be used in discrete units, software as a complex system, and software as an intangible good.

Evidently, software is characterized by a considerable number of economically relevant qualities—ranging from network effects to a subadditive cost function to nonrivalry. Particularly to emphasise is the fact that software fundamentally differs from other information goods: First, from a consumer's perspective the readability and other aspects concerning how the information is presented, is irrelevant. Second, the average consumer/user is interested only in the funtionality of the algorithms but not in the underlying information.

Key words: digital goods, compatibility, information good, network effects, nonrivalry, open source, recombinability, software

JEL-classification: D82, D83, D62, D85, K11

*EMail: Sebastian.Engelhardt@wiwi.uni-jena.de

Contents

Remark

The paper at hand is a *translation* of my 2006 paper "Die ökonomischen Eigenschaften von Software", published in the series Jenaer Schriften zur Wirtschaftswissenschaft, with number 14/2006. Therefore I would like to thank *Antje Russ* (Email: antjeruss@hotmail.com) for the translation. Of course, any mistakes, errors or omissions in this paper are mine.

1 The Object of Study

In the strict sense the term 'software' refers to all non-physical functional components of a computer and thus not only to the computer programs themselves but also to the data which are intended to be processed by the programs (Küchlin & Weber 2005, p 15-16). However, very frequently the term 'software' is used synonymously with 'program' and in contrast to 'data'. The following definition illustrates this specific usage: software is a list of commands and instructions for data-processing (Gröhn 1999, p 4). This paper adopts this distinction between software and data. Possible distinctions like the one between *application* and *system software* (Gröhn 1999, p 5) are not considered.

Software of any kind exists generally in two forms: on the one hand there is the so-called *source program (source code)* and on the other hand the machine-readable *binary code*. This is a direct consequence of the process of programming of software: a program—i.e. the logical structure, the commands and instructions—is written in a specific programming language. The result is a source code which can be read and understood by human beings. In a second step this code is translated into machine-readable form so that the computer can execute the program. This operation is called compiling and the output is the binary code which is—from a semantic perspective—not readable by human beings. To draw conclusions concerning the structure and the programming of the software merely on the basis of the binary code is hardly possible. Accordingly, this paper ignores the possibility of such conclusions about the underlying programming method and also the so-called reverse engineering. Instead it is assumed that the various steps of programming can only be reconstructed when the source code is available (Kooths et al. 2003, p 13-15; Hansen & Neumann 2001, p 155-156).

It is also possible to classify software products according to the rights of ownership respectively exploitation rights and/or different methods of production, that is, one can distinguish between *closed source software (CSS) and open source software (OSS)*. A further distinction between non-commercial OSS, commercial OSS, free-ware, share-ware, and commercial software is also possible (Wichmann & Spiller 2002, p 11). However, it shall not be considered in this paper; for the following explanations it is sufficient to fall

back on the strongly stylized and typified comparison between OSS and CSS. In this the paper follows Pasche & v. Engelhardt (2004), p. 1:

- In the case of OSS the source code and within the underlying programming accomplishment is disclosed. It is permitted to copy, to distribute, to utilize, to modify OSS and to pass on modified versions of OSS (for detailed information see the definition of open source in Open Source Initiative 2004). There are also commercial OSS-products, so-called OSS-distributions. These are coordinated and optimized packages of OSS, and usually services like utility programs, support services and user manuals are added. Although the providers of commercial OSS with their specific products are in competition with one another, they contribute to the same OSS-projects with own developments. This allows cost-sharing and is comparable with the situation in R&D cooperation. (Pasche & v. Engelhardt 2006, p 105).

- CSS on the other hand emphasizes the exclusive use of software. In other words, here the source code, which can be read by human beings, is not disclosed and the software is distributed only in the form of the machine readable binary code. A complete software product is sold but not the underlying programming knowledge which effects the solution. The technical procedure of the compiling prevents the access to the source code. Also, in addition to this, the binary code is equipped with property rights, and the consumer receives the right to use the software by purchasing the corresponding licence (Pasche & v. Engelhardt 2004, p 8). To enforce the excludability, usually a combination of technical and legal copy protection is employed (Kotkamp 2001, p 53).

2 The Economic Characteristics of Software

In order to systematically describe the economically relevant characteristics of the comodity software, the following general definition is used:

> *Software as a complex and recombinant system of commands and instructions for data processing is an intangible good which can be used only in discrete units.*

This definition explicitly emphasizes the various aspects of software as a commodity. In the following, this definition will be used in order to derive the different specific characteristics of software.

2.1 Software as a System for Data Processing

> *Software as a complex and recombinant* ***system*** *of commands and instructions for* ***data processing*** *is an intangible good which can be used only in discrete units.*

The fundamental principle of software (or, the basal sequence of operations in computing in general) is the so-called IPO-principle (Hansen & Neumann 2001, p 652-653): input–processing–output. Whenever data *processing* takes place also data *exchange* occurs—during the data *input* as well as during the data *output*. This becomes obvious when looking at the example of a word processing program: with the help of the keyboard the user 'feeds' the computer with data. If one wants to print the text the user gives a print command. The word processing program interprets this command and sends the data to a printer driver. This printer driver (which is also a software product) translates the received data into the adequate printer language. Afterwards it sends the result of this translation to the printer, which in turn outputs the data in form of printed text on paper. Already this simple example shows three forms of data exchange: the exchange between user and software, between software and software, and between software and hardware. In addition to this, there also exists the possibility of exchange among users: if a file, which was generated with the help of the word processing program, is stored on a disk or USB flash drive, it can be passed on to another user who can modify it if necessary. Here the rule applies "[t]he more users deploy a certain operating system or application software, the easier it is to exchange files" (Kooths et al. 2003, p 18). Thus, the utility of a software product increases with the total number of users of this software, i.e. the size of the network. Software is one of those "products for which the utility that a user derives from consumption of the good increases with the number of other agents consuming the good" (Katz & Shapiro 1985, p 424). Thus, so-called network effects occur and software is a network good (Pasche & v. Engelhardt 2006, p 102). In contrast to physical network goods which are characterized by

physical connections and network nodes software is a *virtual network good* (Fichert 2002, p 2).

According to Blankart & Knieps (1992), p 79, network effects can generally be described with a utility function of the following form:

$$u(x) = u_x + u(N_x) \tag{1}$$

Here x is the network good, u_x is the basic or stand alone utility of the good, and $u(N_x)$ the utility depending on the size of the network N_x. Furthermore $\frac{du(N_x)}{dN_x} > 0$ applies.[1] In the broadest sense of the word there are network goods of which the stand alone utility is zero, for example the telephone or the fax machine.[2] With software, the basic utility can also be zero (e.g., instant messenger programs) but normally one can assume a positive basic utility of software (word processing, spreadsheet analysis). However, one can also imagine software products of which the utility almost completely consist of the stand alone utility and thus the network utility is close to zero (Fichert 2002, p. 4).

In the case of network effects (of which the consumer mostly benefits) the expectations of the consumer can have a great influence on the buying decision: if the consumer expects that a technologically inferior technology b will prevail over technology a, then b has a great advantage over a as far as the expected size of the network is concerned. If the network effect, which is valued according to the expected network size, overcompensates the technical inferiority ($u_a > u_b$), then technology b is actually chosen (Fichert 2002, p 5). This is formally expressed as follows:

$$E\left[u(b)\right] = u_b + E\left[u(N_b)\right] > E\left[u(a)\right] = u_a + E\left[u(N_a)\right] \tag{2}$$

or

$$E\left[u(N_b)\right] - E\left[u(N_a)\right] > u_a - u_b. \tag{3}$$

Thus, in the case of network goods the dominance of inferior technologies over superior technologies is *possible*.

[1] Of course, negative feedback effects are also possible as well. Thus, there might be cases, where the network effect has a inverse U form.

[2] The copy-function of the fax machine shall not be considered.

In the utility function $u(x) = u_x + u(N_x)$ network effects are globally summarized in the term $u(N_x)$. However, one can distinguish between *direct* and *indirect* network effects. (Katz & Shapiro 1985, p 424):

Direct network effects result *directly* from the network characteristics of the good, that is, the utility increases with the distribution of *this* good and the distribution of *compatible* goods respectively (Ehrhardt 2001, p 25). Examples for this are the platforms of Apple and the Wintel standard [3]. As Erhardt points out, the more users actually employ the respective platform, the easier it is for the single user to exchange data with others (Ehrhardt 2001, p 26). The effects of direct network effects are described with 'Metcalfe's Law': if the number of participants n is proportional to the value $u(.)$ of the network or the network good for each participant, the interrelation can be describes as follows: $u(.) \sim n \cdot (n-1)$ (Shapiro & Varian 1999, p 184).

Indirect network effects have basically two causes: first, there is the relevance of *complementary* products and services, and second, there are *learning effects and learning spillovers* (Ehrhardt 2001, p 27-29). The latter refers to the relevance of exchange of knowledge among the participants in a network. The larger the network is and the more users employ the same system software or the same application software, the more likely is it so find somebody who can help you to solve a problem with the software or who can answer your questions about it, and the more attractive is the network good (Kooths et al. 2003, p 18). Such indirect network effects of learning-effects and of learning spillovers can especially be observed in the market for personal computer software (Cowan 1992, p. 291). In the case of new, innovative technologies of which the qualities and possible applications are not yet fully known, information spillovers have another effect: if new technologies, which the consumer can not completely assess, are in competition with one another, it is more attractive for the potential user to buy the technology which has been chosen by the majority of costumers so far. Here, according to Erhardt, the uncertainty concerning the performance of the technology is the lowest because of the numerous adoptions before (Ehrhardt 2001, p 29). The relevance of complementary products and services, i.e., *complementary software*, is very obvious: the more application programs run on an operat-

[3] 'Wintel' is a blending of 'Windows' and 'Intel' and refers to a computer architecture which is also known as 'IBM compatible PC'.

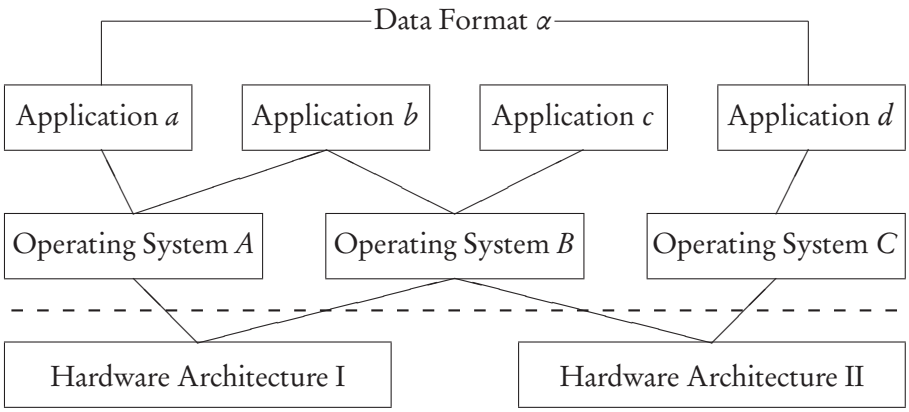
ing system (or are expected to be available in the future), the more attractive it is for the consumer.

The network effects of complementary goods on the demand side are also known as 'hardware-software'-paradigm in network theory (Katz & Shapiro 1985, p 424). This term is derived from the fact that the decision for a certain hardware configuration implicitly limits the set of possible operating systems, for example think about Apple vs. Wintel. Accordingly, Shy (2001) says: "Complementary means that consumers in these markets are shopping for *systems*" (Shy 2001, p 2, emphasis in original). In addition to this, it is the more attractive for *producers* of application software to develop programs for a certain operating system, the more consumers are using this operating system or are expected to use it. This relevance of the (expected) network size for the vendors of complementary products refers to network effects on the *supply side* (for further information on this issue see for example Kooths et al. 2003, p 17). In my opinion these network effects on the supply side can be ascribed to the network effects on the demand side, in other words, on the fact, that these goods form a system. Therefore, in the following I do not distinguish between network effects on the supply side and network effects on the supply side.

The relevance of complementability for (indirect) network effects refers to the relevance of *compatibility*: complementability is usually achieved by the compatibility of the various components (Lopatka & Page 1999, p 955). Compatibility is the fundamental feature of network goods: The decision whether a good belongs to the network x (the system x) or not, depends on the question whether this good is compatible with the other goods of the network x or not. Therefore, compatibility is the central distinguishing mark of different network systems (Katz & Shapiro 1985, p 424 f). This interrelation is presented in figure 1, which shows—in a very general form—the networks respectively the network effects in the area of software. While indirect network effects such as learning-effects and information spillovers are not considered in the figure,[4] the hardware-layer is included for more

[4]This is done because a) the number of users is not depicted and b) the relation between the number of users and the occurrence of *these* indirect network effects is rather trivial: all users, for example of operating system A, are effected by the learning effects and information spillovers related to operating system A. This is very similar to direct network effects.

Figure 1: Substitutes, Complements, Horizontal and Vertical Compatibility



clarity. The application-software a can only be installed on operating system A but application-software b can be installed on operating system A and B. Accordingly, application a and the operating system A are complementary goods; they are vertically compatible. The same holds true for application b and operation system A. Concerning indirect network effects, the operating systems A and B are superior to operating system C because for the first two there are more complementary applications available (two for each: a and b or b and c respectively) than for C. The applications a and d can—as far as direct network effects are concerned (data exchange)—be understood as substitutes: as both applications operate with the same data format α, they are horizontally compatible. The operating systems A and B can also be seen as substitutes when application b is considered to be the network good: on both operating systems the application b can be installed. If one compares the indirect network effects of application b with those of application c, then b is the superior application because there are more complementary goods available for b, in other words, there are the compatible operating systems A and B.

2.2 Software as a System of Commands or Instructions

> *Software as a complex and recombinant* ***system of commands and instructions*** *for data processing is an intangible good which can be used only in discrete units.*

The term software denotes successive commands, i.e. a system of instructions for the processing of data. Therefore, all software can be seen as logic constructs of algorithms and instructions which have to perform one or more tasks. Dosi (1996) distinguishes information from knowledge and explicitly names algorithms and instructions as examples of information (Dosi 1996, p 84). Accordingly, software belongs to the group of information goods. In connection with information goods, that is, whenever the information good is to be traded, the so-called *information paradox* occurs (Kotkamp 2001, p p 55): The buyer of information is not able to determine the value of the information before the transaction unless the vendor reveals the information. However, if the information is revealed, there is no longer a motivation nor a necessity for the buyer to actually purchase the information. Therefore, before the transaction, the information is not or only insufficiently revealed. Consequently, the acquisition of information is always a decision under uncertainty. This explains why

- CSS is solely distributed in the form of the binary code. If the source code was available, nobody would have an incentive to buy the software (in sense of paying a price for the software only, without any service or support etc.). Therefore, there is also no market for OSS itself but only for complementary goods and services. These are, however, often sold together with OSS in so-called OSS distributions (see also section 1).

- consumers as well as the general public are somewhat distrustful of a CSS producer because, due to the concealed source code, the information (the algorithms) remain inaccessible to the consumer even *after* the purchase. For example, in 1999 there was a hot debate about the defragmentation program Diskeeper, which is included in the operating system Windows 2000. The computer magazine c't had reported that Diskeeper was developed by a company named 'Executive Software Inc.' The founder and executive manager of this company, Craig

> Jensen, is a professed scientologist (Göhring 1999, p 58 ff). As such a program has generally direct and active access to all data (Göhring 1999, p 58), this report lead to severe reactions: sect referees called consumers to boycott Windows 2000 (Heise News 2000), in catholic dioceses priests were advised not to use Windows 2000 (Spiegel Online 2000) and the Federal Office for Security in Information Technologies demanded to have access to the source code so that it would be able to evaluate possible dangers caused by the defragmentation program.[5] Moreover, the discussion whether or not the second key 'NSAKey' of the Windows Crypto API is controlled by the US-American intelligence service NSA (cf. Heise News 1999) and the question whether or not Microsoft collects individual-related data unnoticed by the users (e.g. see the critical report about the transfer of a system ID to Microsoft by Persson & Siering 1999, p. 16-18) are expressions of the uncertainty of the consumers. No matter whether or not such accusations and suspicions are correct in particular cases, they are the consequence of an uncertainty of the consumers even after the purchase, which in turn is caused by the fact that the information (the algorithm) remains concealed.

One immediately notices that the commodity software occurs in two distinct variants: either *the information is from the very beginning publicly available information*—we then speak of OSS, which is not traded in a market.[6] Or, the *information is concealed to the costumer before and after the purchase*—we then speak of CSS, which is traded in markets.

Accordingly, software is clearly *different from other information goods*. For example, if you buy a newspaper, the information contained in the newspaper is naturally accessible to the consumer after the purchase. The reason for this difference is that with software, as far as the informational content is concerned, *for consumers*, there is no *relevant* 'utility in addition' because of what embeds the information. This utility in addition because of how the information is presented refers to the following circumstance: For example, the

[5] This has been denied to the German authority. The first governmental office of Europe, which was permitted to view the source code, is the Federal Ministry of the Interior of Austria (Heise News 2001).

[6] To avoid mistakes: OSS distributions and other combinations of OSS with complementary goods *are traded* in markets, but here I refer to OSS *only*.

utility of a newspaper can be divided into the utility derived from the information contained in it and the additional utility derived from the selection and the editing of the information and the readability of the text (the editorial and journalistic service). The existence and relevance of this additional utility of information embedding and presenting are the unique selling proposition of daily newspapers. The great national newspapers differ mostly in their edition and evaluation of contents. The relevance of the utility in addition because of what embeds the information can also be illustrated with the example of scientific texts: two articles about the same topic or with the same thesis may contain exactly the same information. However, because of differences between how the information is embedded/explained the reader may judge/evaluate their quality very differently. This also illustrates why copyright protects the expression but not the underlying information. In the case of professional articles the copyright protects the concrete formulation (expression) but not the underlying discovery (Quah 2003, p 24).

In the case of software, however, such a relevant additional utility for the consumer is missing: possible aspects like an elegant programming, good documentation and a sufficient annotation of the source code are only relevant for software engineers (i.e. the developers, the producers) but not for the users (i.e. the consumers). For the user of the information good software only the 'functioning' of the algorithm, of the technical solution is important and decisive while a possible utility because of the 'expression' is not relevant for the functioning of the software. Thus, software consumers are primarily interested in the *effects* of the algorithms but not in the algorithms themselves. Moreover, a printed version of the source code is mostly useless for users as they lack the knowledge to understand this source code. Accordingly, for the consumers only the fact, that the algorithm of a software product generates a *result*, is important, i.e. the *effects* of software are of interest: a movie is reproduced, a pdf-file can be read etc. This explains why the commodity software creates utility for the consumer *even if the information is not revealed after the purchase*.

As in the case of all information goods, high costs—so-called first copy costs—arise also in the production of software. These costs are also to be considered as sunk costs (Kooths et al. 2003, p 16). In contrast to this, the costs of the reproduction of the complete, digital form of the software-product are rather low. Thus, the crucial production costs are the sunk costs of the

design, the programming, testing procedure etc. while the costs of copying, packing and distribution can be disregarded (see also section 2.6) (Pasche & v. Engelhardt 2004, p 5 f). These first copy costs are a function of the complexity of the software (Bitzer 1997, p 7). For example, the development costs of the operating system Windows 3.1 amounted to about 50 million of US-dollars while the development costs of Windows 2000 were as high as 1 billion of US-dollars (Stelzer 2000, p 837, Arthur1996, p 103, and operating-system.org, 2004). .

When the source code of software is accessible for the general public or for a certain group of users, spillover effects occur which can be interpreted as knowledge spillovers (Pasche & v. Engelhardt 2004, p 18). As the term 'knowledge spillover' includes the term 'knowledge', the terms 'information' and 'knowledge' shall be differentiated. According to Dosi's definition, which is used in this paper, algorithms are information. Therefore, in this article software is considered to be an information good. Knowledge on the other hand includes cognitive categories, rules for the interpretation of information, tacit skills as well as problem-solution and retrieval strategies, which cannot be captured in exact algorithms (Kotkamp 2001, p 30). To illustrate this distinction Dosi (1996) names the example of the argument in mathematics: the argument itself is information but only few mathematicians have the knowledge to understand the argument (Dosi 1996, p 84). However, when in the following explanations knowledge spillovers are mentioned, we refer to the diffusion of information *as well as* the learning and knowledge-related effects on a population which are caused by the diffusion: when an argument is published in a scientific journal it has a number of consequences which go beyond the primary effect of the diffusion of the information itself. By studying the argument some mathematicians may refine their own mathematical skills; some scientists may use the insights of the argument for solutions of other problems etc. *All* these effects of a publication, i.e. the effects on mathematical skills as well as on the diffusion of the information itself, are in the following explanations included in the term *knowledge spillovers*.[7] The publication of a source code is analogous to the publication of an argument. Here also spillovers occur. The effects of the knowledge spillovers are very diverse:

[7]For the interdependence between knowledge and information see (Quah 2003, p 36 f; Loasby 1999, p 148).

in addition to the mere diffusion of the information a general distribution and an enhancement of programming skills within the observed population occur. Furthermore, parts of the source code may be used when new software is developed, i.e. they may be directly adopted and integrated. The last mentioned spillover effect refers to a characteristic of software, which will be discussed in the next section, namely the fact, that software is recombinant.

2.3 Software as a Recombinant System

> *Software as a complex and* ***recombinant*** *system of commands and instructions for data processing is an intangible good which can be used only in discrete units.*

According to Quah (2003), software belongs to the class of *digital goods*. Thus, the following statement holds true also for the commodity which is discussed in this paper: "Digital goods are recombinant. By this I mean they are cumulative and emergent new digital goods that arise from merging antecedents have features absent from the original, parent digital goods" (Quah 2003, p 19). As suggested in section 2.2, elements of programs which already exist can be inserted into other programs. Because of this there are whole catalogues of complete elements of programs (Gröhn 1999, p 5) and a distinct programming approach—the so-called component-based software engineering—emerged. This approach also includes the re-use of software components across producers (Romberg 2003, p 253 ff). Theoretically, there is the possibility to create a new program completely out of elements of already available software products. In other words, it is theoretically possible to *recombine* parts of existing source codes and never write a line of new source code and yet receive a new software product. In practice, however, *available* parts of programs are combined with *new* source code.[8].

Also, the permanent enhancement of software (versions) illustrates the recombinability: out of an old version a new version can be generated by recombining the old source code with new source code elements. This recombinability can be understood as a local spillover. Or, in other words, in the

[8] A popular example for this is the programming environment Delphi of the Borland company: the programming under Delphi is more or less the assignment of attributes to predefined components and the combination of these components. In addition to this, it is also possible to develop new components (Bohne & Lang 2000)

production of software products *economies of scope* are very likely to occur. Economies of scope arise when in the production of n goods production factors can be used, which display the characteristics of collective goods.[9] In the case of software economies of scope arise because the same element of source code can be used for the production of several software products, i.e. n products, without a rivalry in the use of the element of the source code. Thus, source code components are production factors with the characteristics of collective goods.

2.4 Software as a Good Which Can Only Be Used in Discrete Units

> *Software as a complex and recombinant system of commands and instructions for data processing is an intangible good which can be* ***used only in discrete units.***

Software is a good which can only be used in discrete units Quah (2003). On the one hand this means that software is an indivisible good: a software product of which only 50 per cent are copied is not a working program. In contrast to this, 50 per cent of an apple are still half an apple and benefit can be derived form it. A software product is only of use if it is complete, that is, it is beneficial only in integer amounts. On the other hand, software is usually consumed in *one* unit only. Per combination 'consumer and computer' it is sensible to install a certain software product only once; it makes no sense to install it several times. Of course, various consumers can use one and the same computer and therefore, so-called parallel installations may exist. However, these are instances of several consumer-computer combina-

[9] Here the term 'collective good' refers to those characteristics of a production factor which allow its use in several production lines while at the same time its use in one production line does not reduce its use in the other production line (Windisch 1987, p 50). As an example Windisch (1987) names—among others—machines that are not working to full capacity. The term collective good is kept in this paper in order to distinguish *linguistically* between collective goods and the basic forms of the distribution of software—as club goods or public goods—though obviously the terms 'collective goods' and 'public goods' are synonyms.

tions. In general, the consumption of any software product a per consumer-computer combination is always a binary variable:

$$c(a) = x, \qquad \text{with } x \in [0,1] \tag{4}$$

Accordingly, the aggregated demand $\sum_{m=1}^{n} c_m(a) = C(a)$ of n combinations consumer-computer

$$C(a) = X \qquad \text{with } X \in \mathbb{N} \tag{5}$$

is a *discrete* function.

The fact that software can be used only in discrete units and that in the case of software indivisibility prevails refers to the 'fragility' of software . (Quah 2003, p 17). This fragility is due to the intangibility of software: a software program that was correctly transmitted to 99 per cent is useless as software consists of a complex system of commands (figuratively a logic machine) which does not function if the *crucial* command is missing. Software as a complex system will be discussed in the next section.

2.5 Software as a Complex System

> *Software as a* ***complex*** *and recombinant* ***system of commands and instructions*** *for data processing is an intangible good which can be used only in discrete units.*

Software has a very complex structure, consisting of if-then-sequences, logic loops, either/or devices etc. Because of this complexity it is hardly possible to create a program which is completely without mistakes (Graser 2003)—at least if one considers rather comprehensive software products and ignores the more trivial programs like very simple text editors etc. For example, it is assumed that in proprietary software there are in average 0,51 mistakes per one thousand lines of source code (tecChannel News 2003). As the operating system Windows 2000 consists of approximately 30 million of lines (Hochschulrechenzentrum 2003), one can assume that there are about 15.000 mistakes in the source code. Some of these mistakes are never recognized, others lead to program crashes. The fact that software possesses relevant gaps in security, which allow computer viruses to intrude into the computer,

may be interpreted as a special kind of mistake in the source code. One distinguishes between programming mistakes, application mistakes and system mistakes. Programming mistakes in the broadest sense can be described as logic mistakes and slips on the level of the program, i.e. in the interior program structure (Bohne & Lang 2000, p 460-462). Typing errors, so-called infinite loops and periodical fetches, overrun errors or missing releases of resources are examples of this. Application mistakes (user mistakes) on the other hand occur when the user does not use the program 'appropriately', when he makes invalid entries, executes operations in a wrong order or does anything unexpected to the program (Bohne & Lang 2000, p 477). These mistakes still belong to programming mistakes because programs ought to be designed so that the user is guided so far that such misuses can not occur. There are good examples for such a design in practice: for instance, if you enter an ambiguous destination or an illogical date on the online information of the Deutsche Bahn (German Railway Company) you receive the according feedback. Here the system does not permit invalid entries. The last category—system mistakes—concerns the interaction between software and its environment. System mistakes occur for example when the resources of the computer (e.g. working memory) are not sufficient or if peripherals are not accessible (e.g. when a printer is not switched on) (Bohne & Lang 2000, p 482).

It is estimated that the economic costs of defective software in the USA amount up to one per cent of the gross domestic product (NIST & Technology 2002, p ES–2). According to a study of the US-American National Institute of Standards and Technology insufficient quality checks cause annual costs of about 60 million dollar for the US-economy. This study, which concentrates on software developers and users of the automotive industry, the air and space industry as well as the financial sector, concludes that the cost minimizing potential of an improved quality check is about 22 million dollars (NIST & Technology 2002, p ES–11, 1 ff; Heise News 2002). Also the damages caused by computer viruses should not be disregarded. The US-American Computer and Communications Industry Association estimates that the damages caused by the worm SoBig amount to 30 billion US-Dollars and refers to the data of the London-based company mi2g Ltd. which confirms that the "global damage from malicious software inflicted as much as $107 billion in global economic damage" (Bace et al. 2003, p 10).

A consequence of the complexity and defectiveness of software is that a bigger part of the performance of programming consists of troubleshooting and debugging. Some errors become apparent only during the usage of the software by the users. Permanent maintenance is a precondition for a long-lasting use of software products (Franck & Jungwirth 2002, p 125). According to Raymond (1999) about 75 per cent of all costs of a software product in a company are due to maintenance. In addition to the concrete and individual maintenance as a *complementary* service of the producer of software or other providers, the participation in the continuing further development and improvement of a software product with the help of publicly available updates and patches play an important role for the user. The latter mainly takes place via internet downloads. In so far one can assert that software is a good which is not distributed in a final state but which is continually modified and/or enhanced by complementary services, updates and patches.

The fact that some mistakes become apparent only during the use of the software points to another characteristic of software, which can be derived from its complexity: software is a so-called *experience good*[10]—and so in *two* senses (Pasche & v. Engelhardt 2004, p 3; Pasche & v. Engelhardt 2006, p. 101-102):

1. Software is an experience good in the conventional sense (Ewers et al. 2003, p 284 f) because the consumer can only partially assess the characteristics (including possible errors) of a software product before the purchase. The user acquires full knowledge of the benefit only when he practices the use of the purchased software product. Thus, in the case of software the buying decision is always a *decision under uncertainty* (Pasche & v. Engelhardt 2004, p 3). In order to reduce the uncertainty producers of software often offer a 30-day test version to download from their web-sites.

2. Furthermore, the user establishes a *specific form of human capital* by using the software product; from an inexperienced user he or she develops into an *experienced* user. This is the second meaning of experience goods (Pasche & v. Engelhardt 2004, p 3): by using software every day the user becomes better acquainted with the software and extends his

[10]For the term experience good in general see Nelson (1970).

or her abilities in using it. In consequence he or she has 'experience in the usage of the software product *z*' or 'experience in the usage of common office applications', etc. Accordingly, the efficiency in the usage of software increases with the human capital that is generated in this way. For a user without any experiences in the usage of a software product this software can even be useless. This highlights that the utility of software arises in the interplay of software and human capital, which is a good example of Becker's theory of household production (Becker & Michael 1973). This human capital can be generated in a very specific form or a rather broad competence, i.e. on one end of the spectrum there is the knowledge which is related to a certain, highly specific software product and on the other hand there is the general ability to deal with computers (or various operating systems). In this relation software is not different from other tools or machines. Here, too, human capital is generated when the use of these tools or machines is practiced every day and experiences in dealing with them are made (Arrow 1962). This can also be of a rather general or a very specific nature. The same is valid for the fact that tools can be used efficiently only if the users/workers have the necessary knowledge at their disposal. This is even more true, the more complicated and complex a tool or machine is.

The software-specific human capital is a resource (or competence) which can be sold in the labour market. Its significance becomes apparent e.g. in the case of job applications, where software-related requirements are explicitly mentioned, or, when individuals emphazise their software skills in their applications for certain positions. A number of signals have also been established in order to confirm the existence of such human capital. Examples for this are the European Computer Driving Licence (ECDL) or other computer courses where the participant receives a certificate which lists the acquired skill and proficiencies (Pasche & v. Engelhardt 2004, p 3).

In summary, one can establish that:

- the potential utility of software can often be completely assessed only after the purchase (experience good in the first sense),

- the realised utility of using a software product arises out of the interplay between the program and the according human capital,[11]
- the latter is generated by the use of software (experience good in the second sense),
- human capital is a resource or competence which can be sold in the labour market.

2.6 Software as an Intangible Good

> *Software as a complex and recombinant system of commands and instructions for data processing is an* ***intangible*** *good which can be used only in discrete units.*

Software is *intangible* or, in the words of Quah: software is aspatial (Quah 2003, p 18). Software requires a carrier medium, that is a storage medium in order to exist (Cowan & Harison 2001, p 2), but it cannot be identified with the storage medium. For example, a 'CD-ROM of SuSE Linux' or a 'CD-ROM of Windows XP' is merely a CD-ROM on which the respective operating system is stored. More abstractly speaking, the CD-ROM is the *storage medium*, *on which* the intangible good is stored—encoded as a sequence of zeros and ones.

In general, a sequence of zeros and ones can be duplicated or copied as often as one wishes. The result of the copying process—the copy—is *a second original*. This can be compared with the copyability of texts encoded in letters: it can be duplicated and the result is also a second original even it the storage or carrier medium has changed. There may be a text in a book and a photocopied version of the text. One is a hardcover (book), the other one is a number of loose sheets of paper (photocopies) but this does not have any influence on the *content* of the text. The reproducibility of software is analogous to this. However, software is copied digitally and therefore, in general, there is no deterioration of quality—unlike in the analogical process of copying. The *costs* of reproducing software are *nearly zero*, only the costs of the storage media (e.g. DVDs) and the abrasion of the physical duplication technology (e.g. DVD burner) are a cost factor. Quah refers to this characteristic

[11] The same idea in relation to information *in general* can be found in Kotkamp 2001, p. 56.

as the *infinite expandability* of software (Quah 2003, p 13). Consequently, once a software product is developed, it is *not a scarce resource*. Only the storage media (e.g. a CD-ROM) may be subjects of scarcity.

A good which is infinitely expansible cannot be marked by rivalry in use, either (Quah 2003, p 15).[12] This *nonrivalry in consumption* of software prevails on the temporal level as well as on the quantitative level. The intangible commodity software does not wear off, in other words, it is not subject to physical abrasion or consumption (Fichert 2002, p 3; Kooths et al. 2003, p 18). Therefore, an infinite number of individuals may *successively* use a software product. Also in terms of quantity there are no restrictions because a software-product can be reproduced at marginal costs of nearly zero and thus, it can be provided for an infinite number of individuals at the same time.

3 Summary

The insights of this paper may be summarized as follows: Being a network good, software exhibits direct as well as indirect network effects (information spillovers, complementary goods). Consequently, a software product is tendentially the more attractive the more users employ the product. Software belongs to the category of information goods but it differs from other information goods like newspapers, as there is (for the consumer) no utility in addition because of the expression of the information. Moreover, users of software are usually interested only in the functionality of the program but not in the underlying information (algorithm). This phenomenon together with the information paradox explains the twofold distinction between closed source software and open source software. The development of software causes very high costs, the so-called first-copy-costs. In the case of a disclosure of the source code one can expect knowledge spillovers. The recombinability of software leads to economies of scope in the production of software. Software is a good which can be used only in discrete entities and the individual demand for software can be characterized as binary. One can also say that software is distributed in a non-final state: because of the com-

[12] This relation is always valid. However, there are goods which exhibit nonrivalry in consumption but which are *not* infinitely expansible. See also Quah (2003), p 16 on this.

plexity of software it is hardly possible to program a software product which is completely faultless, especially since many errors become apparent only when the program is used. Software is an experience good in two senses: on the one hand the adequacy and the quality can fully be evaluated only after the purchase, in the every day use. On the other hand, the user generates a specific form of human capital while using a software product. A once developed software product is infinitely expansible and is therefore a non-scarce resource. Consequently software is marked by nonrivalry in consumption.

References

Arrow, K. J. (1962), 'The economic implications of learning by doing', *The Review of Economic Studies* pp. 155–173.

Arthur, W. B. (1996), 'Increasing Returns and the New World of Business.', *Harvard Business Review* pp. 100–109.

Bace, R., Geer, D., Gutmann, P., Metzger, P., Pfleeger, C. P., Quarterman, J. S. & Schneier, B. (2003), CyberInsecurity: The Cost of Monopoly. How the Dominance of Microsoft?s Products Poses a Risk to Security, Report, Computer & Communications Industry Association, Washington.

Becker, G. S. & Michael, R. T. (1973), 'On the new theory of consumer behavior', *Swedish Journal of Economics* **75**, 378–396.

Bitzer, J. (1997), The Computer Software Industry in East and West. Do Eastern European Countries Need a Specific Science and Technology Policy?, DIW Diskussionspapiere 149, Deutsches Institut für Wirtschaftsforschung, Berlin.

Blankart, C. B. & Knieps, G. (1992), Netzökonomik., *in* E. Böttcher, ed., 'Ökonomische Systeme Und Ihre Dynamik', Jahrbuch Für Neue Politische Ökonomie, J.C.B. Mohr (Paul Siebeck), Tübingen.

Bohne, A. & Lang, G. (2000), *Go To Delphi 5.*, Addison-Wesley-Longman, München [u.a.].

Cowan, R. (1992), High Technology and the Economics of Standardization., *in* M. Dierkes, ed., 'New Technology At the Outset : Social Forces in the Shaping of Technological Innovations', Campus Verlag, Frankfurt am Main, pp. 279–300.

Cowan, R. & Harison, E. (2001), Protecting the Digital Endeavour. Prospects for Intellectual Property Rights in the Information Society, Research Memoranda 28, MERIT, Maastricht Economic Research Institute on Innovation and Technology, Maastricht.

Dosi, G. (1996), The Contribution of Economic Theory to the Understanding of a Knowledge-based Economy., *in* 'Employment and Growth in the Knowledge-Based Economy', OECD Documents, OECD, Paris, pp. 81–92.

Ehrhardt, M. (2001), *Netzwerkeffekte, Standardisierung und Wettbewerbsstrategie.*, Gabler Edition Wissenschaft : Strategische Unternehmungsführung, Deutscher Universitäts-Verlag, Wiesbaden.

Ewers, H.-J., Fritsch, M. & Wein, T. (2003), *Marktversagen und Wirtschaftspolitik. mikroökonomische Grundlagen staatlichen Handelns*, Vahlen, München.

Fichert, F. (2002), 'Wettbewerbspolitik im digitalen zeitalter. öffnung vermachteter märkte virtueller netzwerkgüter', Beitrag zum 3. Workshop 'Ordnungsökonomik und Recht' des Walter Eucken Instituts.

Franck, E. & Jungwirth, C. (2002), 'Das Open-Source-Phänomen jenseits des Gift-Society-Mythos.', *WiSt - Wirtschaftswissenschaftliches Studium* **31**(3), 124–129.

Göhring, H.-P. (1999), 'Windows 2000 droht ein Bann. Kritik aus den Kirchen an Scientology-Beteiligung', *C't - Magazin Für Computertechnik* **25**, 58–61.

Graser, F. (2003), 'Pragmatischer Umgang mit Softwarefehlern empfohlen', *Computer Zeitung* **18**.

Gröhn, A. (1999), *Netzwerkeffekte und Wettbewerbspolitik. Eine ökonomische Analyse des Softwaremarktes*, Mohr Siebeck, Tübingen.

Hansen, H. R. & Neumann, G. (2001), *Grundlagen betrieblicher Informationsverarbeitung*, Lucius & Lucius, Stuttgart.

Heise News (1999), 'Debatte um NSAKey geht weiter', Online Artikel. www.heise.de/newsticker/meldung/6019.

Heise News (2000), 'Sektenbeauftragter ruft zum Windows-2000-Boykott auf', Online Artikel. www.heise.de/newsticker/meldung/7708.

Heise News (2001), 'Wien darf Windows-Quellcode zuerst prüfen', Online Artikel. www.heise.de/newsticker/meldung/23105.

Heise News (2002), 'Bugs kosten fast sechzig Milliarden Dollar pro Jahr', Online Artikel. www.heise.de/newsticker/meldung/28604.

Hochschulrechenzentrum, U. (2003), 'MS-Windows und Verwandte', Online Artikel. www.hrz.uni-wuppertal.de/dienste/software/os/windows/.

Katz, M. L. & Shapiro, C. (1985), 'Network Externalities, Competition, and Compatibility.', *American Economic Review* **75**(3), 424–440.

Kooths, S., Langenfurth, M. & Kalwey, N. (2003), *Open-Source Software: An Economic Assessment*, Vol. 4 of *MICE Economic Research Studies*, Muenster Institute For Computational Economics, Münster.

Kotkamp, S. (2001), Electronic Publishing. Ökonomische Grundlagen des Handels mit Informationsprodukten, Dissertation, Universität Fridericiana zu Karlsruhe, Karlsruhe.

Küchlin, W. & Weber, A. (2005), *Einführung in die Informatik*, Springer, Berlin Heidelberg.

Loasby, B. J. (1999), *Knowledge, Institutions, and Evolution in Economics.*, Routledge, London.

Lopatka, J. E. & Page, W. H. (1999), Network Externalities., *in* B. Bouckaert & G. De Geest, eds, 'Encyclopedia of Law and Economics', Edward Elgar Publishing Limited, Cheltenham [u.a.], pp. 952–980.

Nelson, P. (1970), 'Information and Consumer Behavior.', *Journal of Political Economy* **78**(2), 311–329.

NIST, T. N. I. o. S. & Technology, eds (2002), *The Economic Impacts of Inadequate Infrastructure for Software Testing. Final Report*, Planning Report.

Open Source Initiative (2004), 'The Open Source Definition', Online Dokumrnt. www.opensource.org/docs/definition_plain.php.

operating-system.org (2004), 'Windows Family', Online Artikel. www.operating-system.org/betriebssystem/_german/bs-windows.htm.

Pasche, M. & v. Engelhardt, S. (2004), Volkswirtschaftliche Aspekte der Open-Source-Softwareentwicklung., Jenaer Schriften Zur Wirtschaftswissenschaft 18/2004, Friedrich Schiller Universität, Jena.

Pasche, M. & v. Engelhardt, S. (2006), Führt Open-Source-Software zu ineffizienten Märkten?, *in* B. Lutterbeck, R. A. Gehring & M. Bärwolff, eds, 'Open-Source-Jahrbuch 2006', pp. 93–108.

Persson, C. & Siering, P. (1999), 'Big Brother Bill. Microsofts heimliche ID-Nummern - angeblich eine Panne', *C't - Magazin Für Computertechnik* **6**, 16–20.

Quah, D. (2003), Digital Goods and the New Economy., CEP Discussion Papers 563, London School of Economics, London.

Raymond, E. S. (1999), 'The Magic Cauldron', Paper. www.catb.org/ esr/writings/magic-cauldron/magic-cauldron.ps.

Romberg, T. (2003), Herstellerübergreifende Wiederverwendung von Komponenten., *in* 'Handbuch Zur Komponentenbasierten Softwareentwicklung', -.

Shapiro, C. & Varian, H. R. (1999), *Information Rules. a strategic guide to the network economy*, Harvard Business School Press, Boston.

Shy, O. (2001), *The Economics of Network Industries.*, Cambridge University Press, Cambridge.

Spiegel Online (2000), 'Katholiken misstrauen Scientology-Software', Online Artikel. www.spiegel.de/netzwelt/politik/0,1518,79730,00.html.

Stelzer, D. (2000), 'Digitale güter und ihre Bedeutung in der Internet-ökonomie', *WiSt - Wirtschaftswissenschaftliches Studium* **6**, 835–842.

tecChannel News (2003), 'Apache-Code ist kommerziellen Produkten ebenbürtig', Online Artikel. www.tecchannel.de/news/20030702/thema20030702-11120.html.

v. Engelhardt, S. (2006), Die ökonomischen Eigenschaften von Software, Jenaer Schriften Zur Wirtschaftswissenschaft 14/2006, Friedrich Schiller Universität, Jena.

Wichmann, T. & Spiller, D. (2002), Free/libre and open source software: Survey and study – final report, Part 3: Basics of open source software markets and business models, Berlecon Reasearch, Berlin.

Windisch, R. (1987), Privatisierung natürlicher Monopole. Theoretische Grundlagen und Kriterien, *in* R. Windisch, ed., 'Privatisierung Natürlicher Monopole Im Bereich von Bahn, Post Und Telekommunikation', Mohr Siebeck, Tübingen, pp. 1–146.