Chapter 14
Software

14.1 The Software Market

"Software" refers to programs that can be run on hardware (generally a computer). These programs are in code. The market for software (Mowery, 1995) is distinguished by virtue of its particular complexity: there is not only a wealth of applications, but also a multitude of respective new versions following each other chronologically or variants, offered at the same time, which differ in terms of their functionalities. For many programs, it is important that they be attuned to each other. To comprehensively describe the products of such a market in the context of a single chapter does not appear possible, which is why we will content ourselves with a general overview and concentrate on describing the creation of software.

As complex as the market is in terms of products, it is extremely simple from the companies' perspective. For commercial software, there are a select few companies who dominate–we need only consider Microsoft's monopoly status. However, in the single market segments, too, there is often a single company that "calls the tune". Thus, there is a clear market leader, in SAP, for programs of Enterprise Resource Planning (ERP); the market for database system is dominated by Oracle. Apart from commercial products, we can find Open Source software–which is just as highly developed–such as Linux' or Apache's, which is the product of voluntary (and unpaid) cooperation between software developers. From a user perspective, the choices represented here are not summed up as "either/or", but increasingly as "not only but also", since products from both worlds are often interoperable (Baird, 2008).

Dominant companies, and dominant products within market segments, provide for a high functionality in the programs, but also to a great vulnerability, since standard programs in particular are susceptible to attack (this is where the criminal's work investment "pays off"). Among the quality criteria for software are thus both optimal functionality and an equally optimal software security.

In a first rough classification (Buxmann et al., 2008, 4), we can distinguish **system software** (e.g. operating systems, network software or programming languages), **machine-oriented software** such as Middleware ("connecting" software, which allows programs to interact) and database and **application software** (e.g.

retrieval systems). Within application software, we also differentiate **individual software** (which is "tailor-made" for a specific task in a company) and **standard software**, which is produced for the mass market. For the latter, we classify via the kind of usage, and are left with software for commercial usage, such as ERP or knowledge management systems (with a multitude of programs, such as systems for document, project, customer relationship or customer knowledge management) (Gust von Loh, 2009), software for commercial and private usage (browsers, office software) and software for purely private implementation (such as games or software for looking at and editing pictures). Figure 14.1 collects all these aspects in one classification.

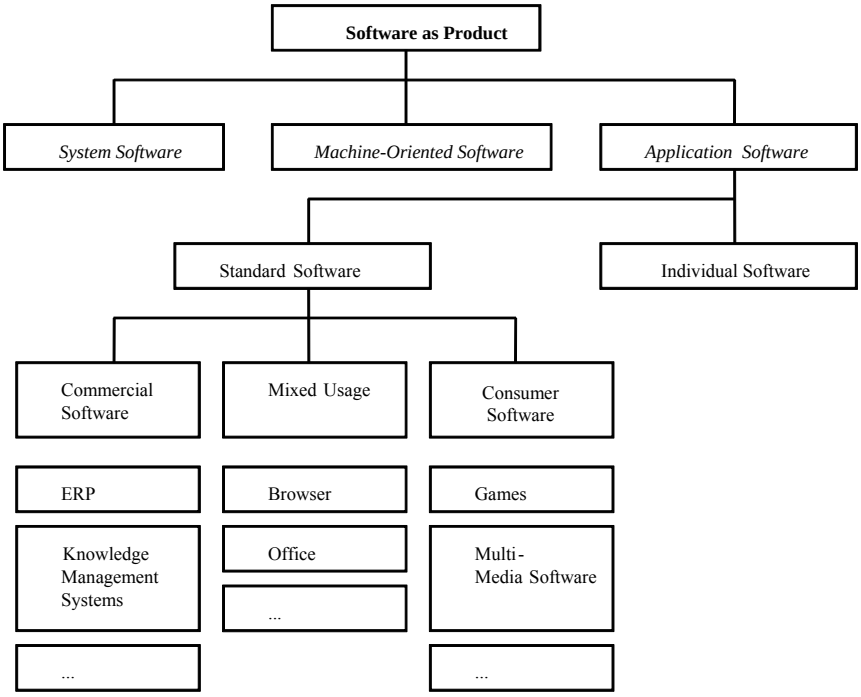

Figure 14.1: Rough Classification of Software.

The software products are joined by software services. Here, we distinguish between consulting and implementation services and the operation of application software as a service. **Consulting and implementation services** are often necessary when there is insecurity concerning the kind of software to be used or when the required software is difficult to implement in the company (Buxmann et al.,

2008, 7). Such service providers appear in the form of IT service companies, system integrators or systems houses. Some consulting services also specialize in software selection and implementation.

Certain companies decline to acquire application software and use it in-house, instead outsourcing this operational procedure to third parties. Such companies host application software and offer their services on a subscription basis. Here, we speak of **Software as a Service** (SaaS) (Buxmann et al., 2008, 8et seq.). Figure 14.2 shows our classification of software services.

Software companies generate their revenue either by selling licenses for their products or by offering services (or from both areas).

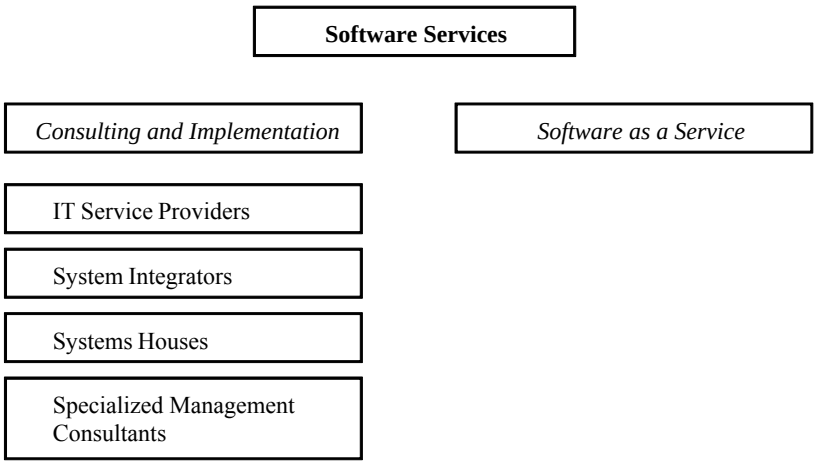

Figure 14.2: Rough Classification of Software Services.

14.2 Software Development

Depending on our starting point, we distinguish between five kinds of software development (the first three following Ruparelia, 2010, 12):

- on the basis of specifications (Cascade model, b-model, V-model),
- on the basis of risk (to be avoided) (spiral model),
- on the basis of concrete scenarios (simplified model),
- on the basis of the development process (agile software development),
- component-based development (can be combined with one of the above methods).

Every software development must be both effective and efficient (Zave, 1984, 112-113). Effectiveness ("are we doing the right things?")–called "Validation (Building the Right System)" by Zave–is demonstrated by customers successfully implementing the software in solving their problems, where the users are familiar with the intended applications, but not with computer systems. Efficiency ("are we doing things the right way?"), or "Verification (Building the System Right)", means that the system thus created fulfils the formulated expectations and specifications, but it also means that the (financial or personal) means applied during production have been used ideally.

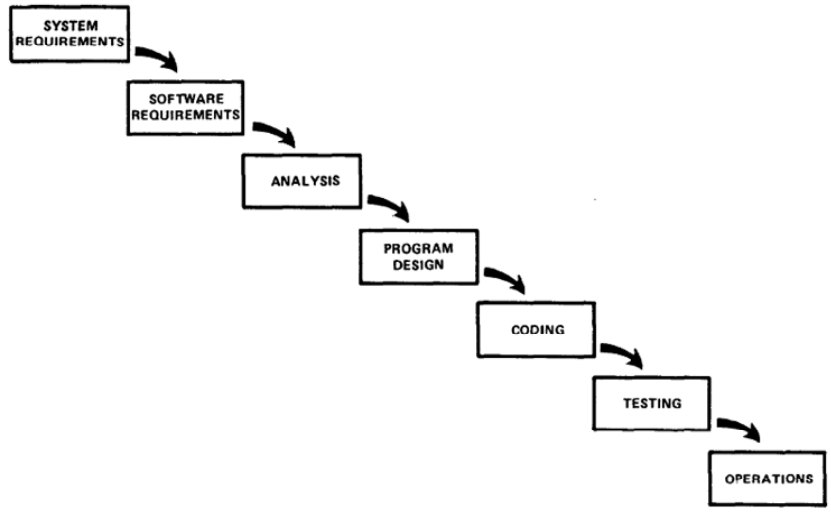

Figure 14.3: The Cascade Model of Software Development. Source: Royce, 1970, 329.

At the beginning of the traditional models of software development are the specifications, in other words, what the system to be created is expected to offer in terms of functionality. As early as 1956, Benington introduced a corresponding model (Benington, 1987), which was fleshed out by Royce into the **Cascade model** in 1970 (Figure 14.3). The way from the requirements to the working system proceeds via several stages, which much each be planned and staffed. On the basis of the specifications for the entire system (which also comprises hardware), the software specifications are separated and analyzed in such a way that they become programm*able*. Only after the program's design has been conceived does the actual programming ("coding") of the desired solution begin. This is then tested extensively and, in case of positive results, released. One should not imagine this to be a linear and one-track process, however. Rather, there is a feedback to previous stages every step of the way. Of particular importance is the interplay of soft-

ware specifications and program design, since it is decided only during the conception of the software whether the requirements even make sense and, particularly, if they are realizable. A further central feedback loop is located between testing and product design, since here it is shown how the design "runs" during operation. Royce (1970, 332) emphasizes the importance of project documentation, since detailed notes on the acquired status of the project are necessary at every step of the software development. To avoid errors in the product, Royce (1970, 334) recommends repeating the entire process ("do it twice"), the goal being, firstly, the prototype, and the operative product second. In conclusion, Royce (1970, 338) sets out five "golden rules" of software development:

> Complete program design before analysis and coding begins.
>
> Documentation must be current and complete.
>
> Do the job twice if possible.
>
> Testing must be planned, controlled and monitored.
>
> Involve the customer.

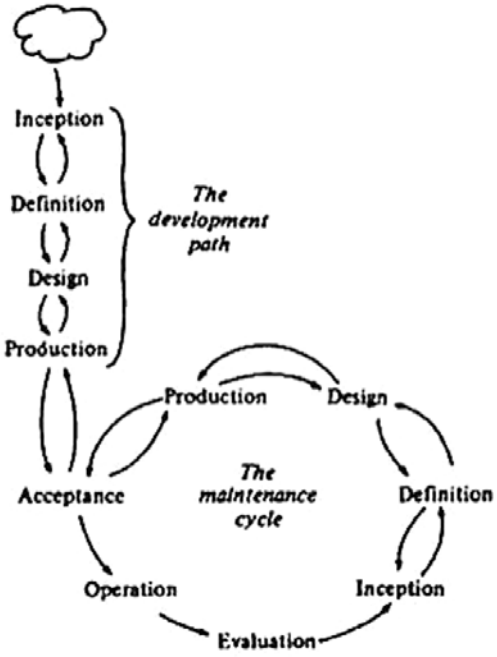

Figure 14.4: The b-Model of Software Development. Source: Birrell & Ould, 1985, 4.

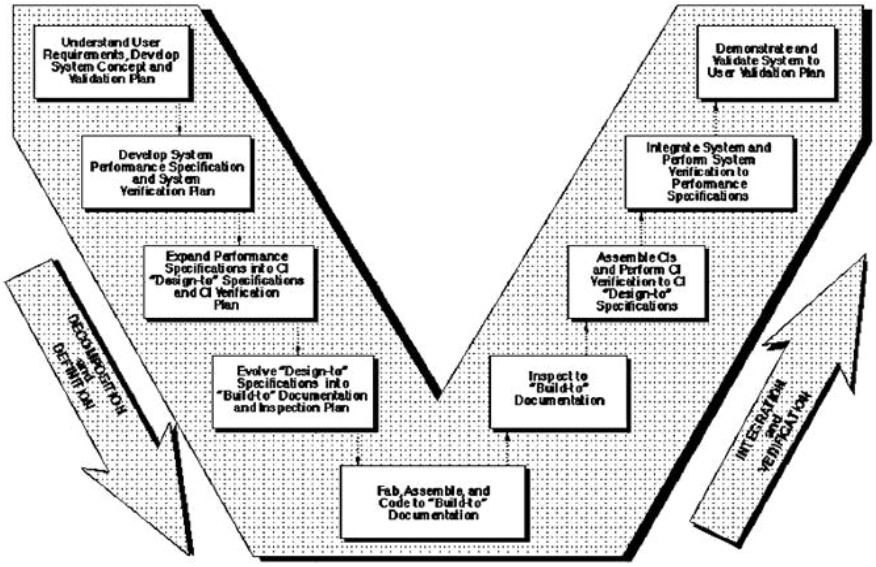

Figure 14.5: The V-Model of Software Development. Source: Forsberg & Mooz, 1995, 5. CI = Configuration Item.

Birrel and Ould (1985) split the production process in two fundamental stages in their **b-model** (Figure 14.4). The first stage–the development path–is laid out analogously to the Waterfall model. Birrel and Ould emphasize that a software is never "final", but requires constant maintenance and further development. In this respect, the second stage–the maintenance cycle–must be heeded, leading as it does to a sequence of versions of the original software.

The **V-model** by Forsberg and Mooz (1995), as used by NASA, also follows the Cascade model at first, but splits the overall process into two subphases (Figure 14.5). The process starts at the top left, with the users' information requirements, and ends up at the top right, with the information system as accepted by the user. On the left-hand side of the V, user specifications are disassembled into "configuration items" and defined as precisely as possible, in order to be assembled in an integrated way–as software items–on the right-hand side. Here, the single (horizontal) levels correspond to each other: user requirements are opposed by the system, as positively evaluated by the users (topmost level), the system architecture by the integrated system and the design work corresponds with the system integration work (levels 3 and 4), so that comparisons between the different stages of the requirements (left-hand side) and the stages of system development (right-hand side) can be made at any given time.

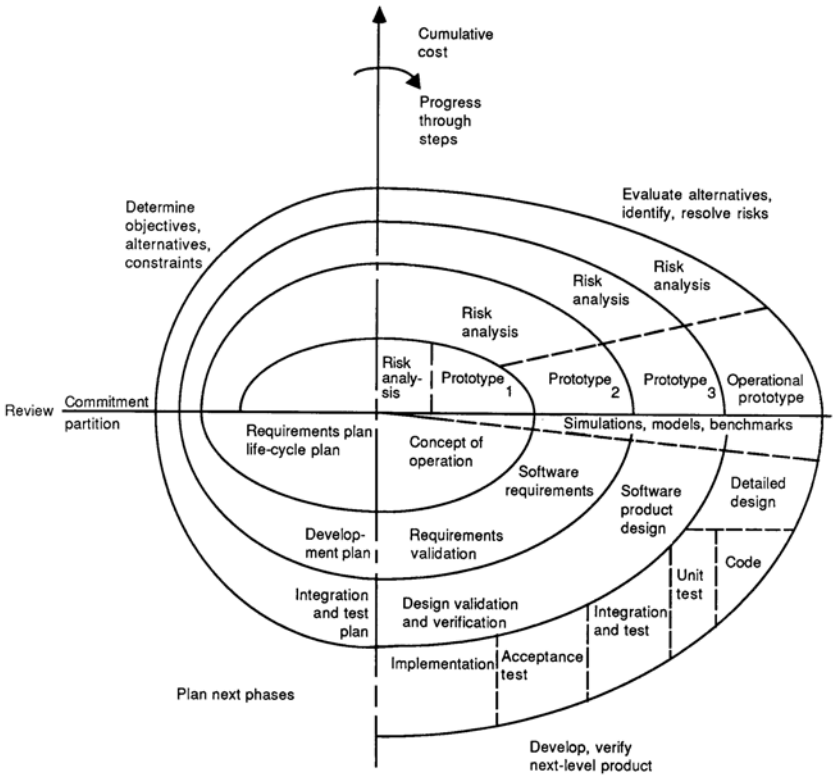

Figure 14.6: The Spiral Model of Software Development. Source: Boehm, 1988, 64.

Software development is an expensive and risky business. Boehm's **spiral model** (1988) (Figure 14.6) always keeps this risk in mind, being characterizable via "start small, think big" (Ruparelia, 2010, 10). The elements of the Waterfall are still granted great importance, but they are no longer run through in their entirety at the beginning. The Cascade model's top-down approach is replaced by a look-ahead perspective. The first prototype is the result of a feasibility study and is thus very primitive, but it is meant to make it possible to estimate whether the risk of starting the project is worth it in the first place. In the second run, the requirements are specified and analyzed. At the end of this run, there is again a prototype, which is subjected to a risk analysis. Bit by bit–secured via a risk analysis after each round–an operative prototype is developed, which can be fleshed out into a product. The great advantage of the spiral model is its continuous risk estimate, and thus its cost control of software development

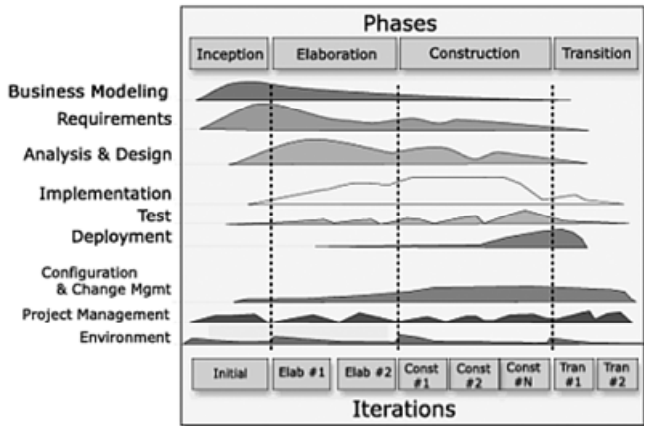

Figure 14.7: The Unified Model of Software Development. Source: Jacobson et al., 1999 and Intelligent Systems.

The **unified model** by Jacobson, Booch and Rumbaugh (1999) starts with the concrete case of software development and unifies aspects of both the Cascade and the spiral model (Figure 14.7). Within the four phases (beginning, conceptional elaboration, software construction and transition to the market phase), several rounds of iteration are run through, as in the spiral model. Differently weighted according to the phase, the objective is to run through six core disciplines of software development: moulding the business model, specifications, analysis and design, implementation, testing and practical application. Here we recognize the building blocks of the Cascade model without any difficulty. Complementing the core process, attention is also granted to accompanying activities such as change management or project management.

The approaches to software development sketched thus far can be summarized as being "plan-driven"–they pursue an elaborate plan and document every step. Not so the "light-weight" methods, such as the Dynamic Systems Development Method, Feature-Driven Development or Extreme Programming, which, put together, we call **agile software development**. This method is distinguished via a non-linear process, in which frequent, short feedback loops occur between developers, among each other, and between developers and customers, in the sense of "inspect-and-adapt" (Williams & Cockburn, 2003, 40). The "manifest of agile software development" formulates four fundamental behavioral guidelines:

> individuals and interactions over processes and tools,
>
> working software over comprehensive documentation,
>
> customer collaboration over contract negotiation,
>
> responding to change over following a plan.

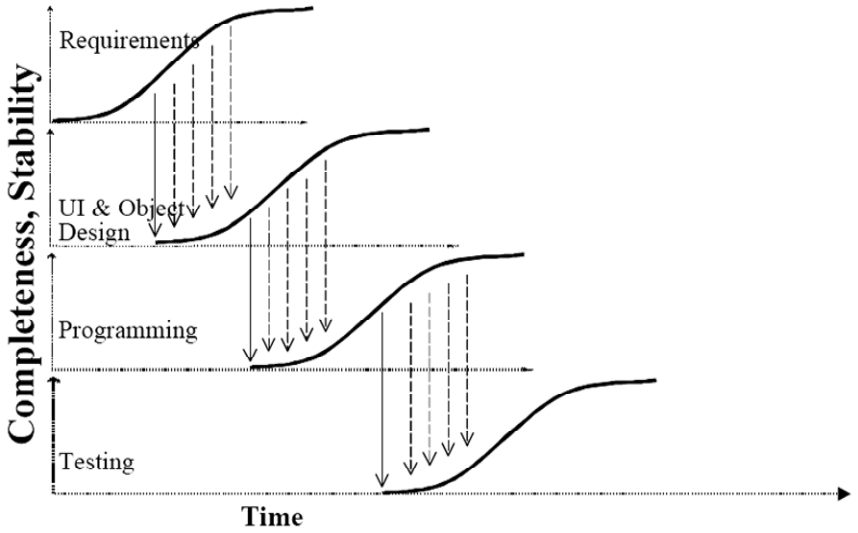

Figure 14.8: Agile Software Development with Overlapping Project Phases. Source: Cockburn, 2000.

One orients oneself more on people and communication than on set plans in project management (Cockburn, 2000, 8). Communication itself–since it is always less than perfect–must be guided. This is how software development becomes a game, which is played in a team and pursues goals. Alistair Cockburn (2000, 33 and 40) describes agile software development as a "goal-directed cooperative game" and as a "game of invention and communication". The group of developers starts their work as early as possible, so that project phases, which are normally worked through one after the other, overlap. Here it is essential for the information of each previous stage to be constantly updated (indicated in Figure 14.8 via the dashed arrows). Updates are made via direct communication and not via written documentation. This is expressed particularly clearly in Extreme Programming (XP): we deliver software, not documentation (Cockburn, 2000, 141).

Since it is dependent on direct communication, agile software development is suited for small teams (less than 50 developers) and companies that are not certified according to the quality management norm ISO 9000, because ISO 9000 prescribes strict documentation. However, it is possible to combine agile software development with one of the plan-based methods (Boehm & Turner, 2003).

Software consists of components–for instance, a text processing software will have the integrated components of spellchecking or hyphenation (Brereton & Budgen, 2000). It is advisable to use such components multiple times and incorpo-

rate them into systems. This is the basic idea of **component-based software development**, which dates back to McIlroy (1969). Component-based development can be combined with any of the previously introduced models of software production.

What does software development look like in practice? We will briefly sketch this on the example of **Microsoft**. There are loosely linked small teams of developers, who frequently synchronize their work results and stabilize the product in development. Added to this are continuous tests of the software. This "Synch-and-Stabilize" approach (Cusumano & Selby, 1997, 54) knows different project phases (planning, development, stabilization), but it does not run through the stages of the Cascade model one after the other, instead pursuing an interative approach. Cusumano and Selby (1997, 55) report:

> The waterfall model has gradually lost favor, ..., because companies usually build better products if they can change specifications and designs, get feedback from customers, and continually test components as the products are evolving. As a result, a growing number of companies in software and other industries–including Microsoft–now follow a process that iterates among design, building components, and testing, and also overlaps these phases and contains more interactions with customers during development.

Thus it can definitely happen that more than 30% of specifications in the planning phase are amended during later development stages (Cusumano & Selby, 1997, 56). The products are offered on the market as long as they are "good enough". In other words, one does not wait until something becomes "perfect" (Cusumano & Selby, 1997, 60).

14.3 Globalization and "Offshoring"

The software industry is aligned internationally. Programs can–at least in principle–be developed anywhere, with transport costs, in contrast to the value chain of physical goods, being negligible. The **globalization** of this industry is not only of importance for the labor markets, but also for distribution. There are hardly any national "home markets" for software; rather, software can be sold the world over (Buxmann et al., 2008, 156 et seq.).

If we want to make the international buying and labor markets usable for software production, we must decide whether to found subsidiaries abroad (or enter joint ventures with domestic enterprises) or whether to contract a third party. The latter method is called–no matter what country is concerned–"outsourcing". Outsourcing activities abroad is either "nearshoring", when the countries are close by (from the U.S.A.'s perspective Canada or Mexico, from the German perspective the Czech Republic, Poland, Hungary and Slovakia), or "offshoring", when far-

away countries are concerned (such as India, for companies with their seat in Germany or the U.S.). Table 14.1 summarizes these definitions.

Outsourcing ***Nearshoring*** ***Offshoring***			*Contractee has its seat...*	
		domestically	*in neighboring countries*	*overseas*
Shifting of internal activities to...	*associated enterprises*	---	Nearshoring without Outsourcing	Offshoring without Outsourcing
	foreign enterprises	Outsourcing	Nearshoring with Outsourcing	Offshoring with Outsourcing

Table 14.1: Systematization of Outsourcing, Nearshoring and Offshoring. Source: Following Mertens et al., 2005, 2.

The shifting of internal activities abroad without outsourcing means the founding of subsidiaries or entering joint ventures with domestic companies. The goals are cost savings via lower salaries in the target country as well as the option of tapping the respective foreign markets. Since the creation of one's own subsidiary "from the bottom up" in an unknown country requires a lot of effort, joint ventures with established enterprises from the target country allow a company to profit from their knowledge of the country and preliminary work. Here the difference between nearshoring and offshoring becomes clear. In nearshoring, the cultural (but also the temporal) distance is far shorter than in offshoring, which means that subsidiaries make more sense in closer proximity. The shifting of one's activities to foreign companies, i.e. outsourcing, can be done domestically or aim for close-by countries (with similar cultures) or far-off countries (with the disadvantage of cultural differences). The software industry makes use of nearshoring, offshoring and outsourcing like few other branches of the industry. India in particular has become an important exporter of software and partner of foreign software companies.

What are the motives that lead software companies to practice outsourcing as well as nearshoring/offshoring? Buxmann, Diefenbach and Hess (2009, 165 et seq.) detect five bundles of motives:

- Cost savings (lower salaries in the nearshore and offshore locations, but connected to a higher coordination effort–particulary offshore),
- Higher flexibility (in outsourcing, services can be bought precisely when needed, thus reducing one's own fixed costs),
- Concentration on core competencies (shifting more peripheral activities abroad while dealing with the important aspects oneself),

- Acquiring know-how (India in particular has many and well-trained information specialists, which are not available in the national labor markets of, for instance, Germany and the U.S.–in such numbers at least),
- "Follow the Sun" (Development and Service Centers intelligently placed around the world allow for service around the clock, due to the different time zones).

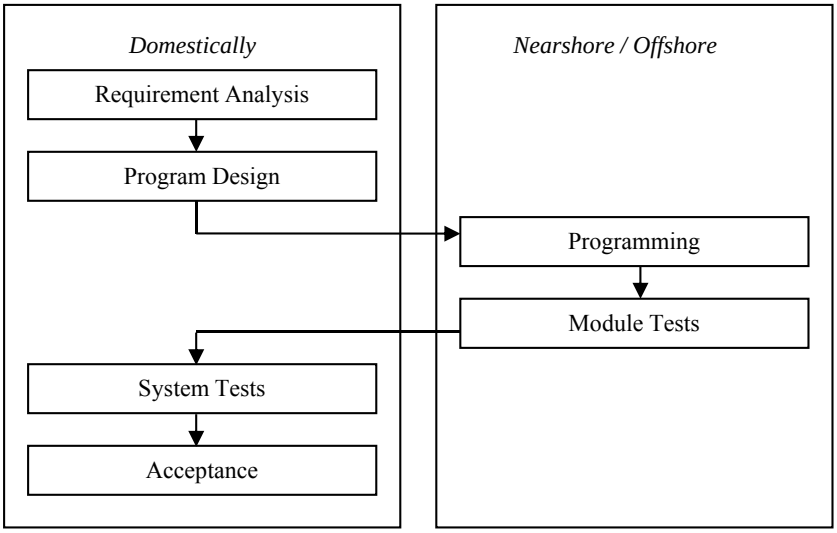

Figure 14.9: Phases of Software Development Domestically and Nearshore/Offshore. Source: Following Buxmann et al., 2009, 178.

If we regard the steps of software development, we can see that not all stages of the creation process are suitable for nearshoring or offshoring. Buxmann et al. (2009, 178) discuss the option of preferentially shifting abroad routine tasks such as programming (following detailed specifications) and tests of the programmed modules, while keeping the other steps in-company (Figure 14.9).

What are the effects of globalization on **SAP**? This company, based in Walldorf, Germany, produces business software and is the worldwide market leader within this segment (Schuster et al., 2009). The stage of requirement analysis is distributed internationally by SAP (to subsidiaries as well as independent companies), since proximity to the respective customers allows the company to meet an optimum of specific requirements. The rough planning for the project is done in Walldorf, while the concrete program design, programming and testing are done in SAP's development centers, scattered around the world (Schuster et al., 2009,

191). Apart from various smaller development centers, SAP keeps four large centers: (in order of their strategic importance) in Walldorf, Bangalore, Montreal and Palo Alto. As far as it does not touch upon highly sensitive areas, programming can be shifted to India. For software tests, SAP keeps a test team in Pune (India) (Buxmann et al., 2008, 180-181.). This distributed processing results in the project teams' high creativity level, due to employees' different cultural backgrounds and–via the "Follow the Sun" principle–project working times of 24 hours every day. The headquarters in Walldorf supervises the process of the decentralized activities and integrates the individual work packages. The software is implemented on location, by the customer. Customer service and system support are guaranteed around the clock by three call centers in Walldorf, Philadelphia and Singapore. Here, "Follow the Sun" is essential, as Schuster, Holtbrügge and Heidenreich (2009, 192) report:

> Since SAP often supports all of a company's business processes, such a company will be unable to operate in case of system failure, which makes around-the-clock service availability a decisive competition factor for the customer.

14.4 Conclusion

Only available in the printed version.

14.5 Bibliography

Baird, S.A. (2008). The heterogeneous world of proprietary and open-source software. Proceedings of the 2nd International Conference on Theory and Practice of Electronic Governance (pp. 232-238). New York, NY: ACM.

Benington, H.D. (1987). Production of large computer programs. Proceedings of the 9th International Conference on Software Engineering (pp. 299-310). Los Alamitos, CA: IEEE Computer Society Press.

Birrell, N.D., & Ould, M.A. (1985). A Practical Handbook to Software Development. New York, NY: Cambridge University Press.

Boehm, B.W. (1988). A spiral model of software development and enhancement. Computer / IEEE, Sept., 61-72.

Boehm, B., & Turner, R. (2003). Using risk to balance agile and plan-driven-methods. Computer / IEEE, 36(6), 57-66.

Brereton, P., & Budgen D. (2000). Component-based systems. A classification of issues. Computer / IEEE, Nov., 54-62.

Buxmann, P., Diefenbach, H., & Hess, T. (2008). Die Softwareindustrie. Ökonomische Prinzipien, Strategien, Perspektiven. Berlin, Heidelberg: Springer.

Cockburn, A. (2000). Agile Software Development. (Online).

Cusumano, M.A., & Selby, R.W. (1997). How Microsoft builds software. Communications of the ACM, 40(6), 53-61.

Forsberg, K., & Mooz, H. (1995). The Relationship of System Engineering to the Project Cycle. Cupertino, CA: Center for Systems Management.

Gust von Loh, S. (2009). Evidenzbasiertes Wissensmanagement. Wiesbaden: Gabler.

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). The Unified Software Development Process. Reading, MA: Addison-Wesley.

McIlroy, M.D. (1969). Mass produced software components. Naur, P., & Randell, B. (ed.), Software Engineering. Report on a Conference Sponsored by the NATO Science Committee (pp. 138-155). Garmisch, Germany, 7th to 11th October 1968. Brussels: NATO.

Mertens, P., Große-Wilde, J., & Wilkens, I. (2005). Die (Aus-)Wanderung der Softwareproduktion. Eine Zwischenbilanz. Erlangen, Nürnberg: Friedrich-Alexander-Universität. (Arbeitsberichte des Instituts für Informatik. Friedrich-Alexander-Universität Erlangen Nürnberg; 38,3).

Mowery, D.C. (1995). International Computer Software Industry. New York, NY: Oxford University Press.

Royce, W.W. (1970). Managing the development of large software systems. Proceedings of the 9th International Conference on Software Engineering (pp. 328-338). Los Alamitos, CA: IEEE Computer Society Press.

Ruparelia, N.B. (2010). Software development lifecycle models. ACM SIGSOFT Software Engineering Notes, 35(3), 8-13.

Schuster, T., Holtbrügge, D., & Heidenreich, S. (2009). Konfiguration und Koordination von Unternehmungen in der Softwarebranche. Das Beispiel SAP. In Holtbrügge, D., Holzmüller, H.H., & von Wangenheim, F. (eds.), Management internationaler Dienstleistungen mit 3K. Konfiguration–Koordination–Kundenintegration (pp. 174-202). Wiesbaden: Gabler.

Williams, L., & Cockburn, A. (2003). Agile software development: It`s about feedback and change. Computer / IEEE, 36(6), 39-43.

Zave, P. (1984). The operational versus the conventional approach to software development. Communications of the ACM, 27(2), 104-118.