The Tension between Integration and Fragmentation in a Component Based Software Development Ecosystem

Jens-Magnus Arndt, Jens Dibbern
University of Mannheim, Germany
{jens.arndt|dibbern}@uni-mannheim.de

Abstract

Historically, software application systems have been produced either individually tailored for the specific customer, or they have been adapted from standardized packages. This paper proposes that component based software development could provide a synthesis of both approaches, combining the best aspects of each of the other two paradigms. Taking a closer look at the underlying principles of software reuse and modularity, however, reveals a number of conflicting forces that influence the acceptance and diffusion of this new paradigm within and throughout the software industry. A dialectical change perspective is used to explain this change process. It is argued that traditional software producers have a dominant position in their industry, and will be able to retain this position for some time. However, the concept of mindful innovation is introduced to give an outlook how smaller software producers might exploit the opportunities that this paradigm shift provides.

1. Introduction

Taking a historic perspective on information systems (IS) development, there have been two prevalent paradigms: the individually tailored, custom made solution and the off-the-shelf, standardized, pre-packaged solution. Each paradigm has advantages and disadvantages, so that – contrary to Kuhn's [24] definition of a paradigm – both approaches coexist in practice up to now. However, recent years have shown the uprising of a different paradigm – the approach of component based software development (CBSD).

This paper provides a brief historical analysis of both opposing paradigms, and – using a dialectical change perspective [41] – describes the new paradigm as emerging as a synthesis of the other two. A discussion of the current literature on CBSD identifies two underlying theories of the concept – software reuse and modularity. Reviewing these theories, it is argued that the actual novelty in the concept can be seen in the proposed fragmentation of component development across organizational boundaries. While software has been developed within one organization in the two preceding paradigms, establishing a marketplace in which components can be traded is a crucial part of the new paradigm. This leads to the application of transaction cost theory [45], and through this, towards conflicting prognoses of the development of the industry of software development. Finally, the recently adopted concept of mindfulness [39] in the IS field is proposed as one possible answer to these conflicting prognoses.

2. IS development from a historic perspective

When the first computers were sold to corporations during the 1950s, there was hardly any market for commercial software – the term software itself had not even been coined until 1959 [11]. In this environment, there were three distinct ways of acquiring the needed computer programs: they were bundled with the hardware; they were exchanged freely through communities of interest; or they were developed in-house by hired computer staff [11].

In this context, a demand for 'custom' programs emerged – programs that were not supplied by the hardware vendor, that could not be acquired through a community of interest, and that could not or were not intended to be produced in-house. This market opportunity has been seized by *software contractors*, companies that supply exactly these computer programs. In 1965, there were about forty to fifty large, and an uncountable number of small contractors that developed individually designed software for corporations [11, 22].

However, this individually designed nature of the computer programs changed – albeit slowly – during the late 1960s and the early 1970s. A reason for this can be found in the rapid development of technology as opposed to the rather slow development in software productivity (often referred to as the *software crises*). Campbell-Kelly [11] points out that during the 1960s, computer performance has increased by two orders of magnitude, while software productivity has only increased by a factor of two to three. Consequently, there has been a growing

gap between the computing power, and the capability of custom made software used to exploit it. It has become less feasible for companies to develop their own software. This, in combination with the fact that under growing pressure of the U.S. Justice Department [14], IBM decided in 1969 to unbundled its hardware from its software [22], has persuaded several software contractors to convert their existing software into pre-packaged standard software [11]. Despite the fact that in 1970 still an overwhelming majority of software spending has been dedicated to software contracting, the development towards the use of standardized software packages that started in this time could not be stopped.

To illustrate this development, fast-forward to the turn of the millennium, and you find a multitude of different research efforts that all indicate that standardized, off-the-shelf systems are the norm and that it is – albeit possible [9] – a rather unwise decision to modify the standard beyond the predefined configuration options. It at best increases the time and money that is required to get a system up and running, at worst it might have severe consequences for the whole organization [37]. Yet, despite the current dominance of standardized IS, in certain situations, individually designed software is still used in organizations. Fuggetta [20], for example, mentions governmental agencies as a prime target for custom-made software.

In other contexts, this decision is addressed as part of the *make-or-buy decision*. This terminology however, is slightly misleading. While the make-or buy-decision should address the fact whether software is produced in-house or bought from a third party, the here addressed issue revolves around the distinction whether a piece of software is written individually for a customer or for a mass market. Nevertheless, the literature on the make-or-buy decision covers some of the aspects that are important for this research. Szyperski [40], for example, also uses the term make-or-buy. However, he very well illustrates the differences between the two paradigms introduced above: customized software offers a high degree of flexibility, a high fit with the business needs, and consequently gives the opportunity to differentiate the organization from its competitors. In contrast, standardized software is usually more cost efficient and faster when it comes to the roll-out, and it is easier to maintain and upgrade – an activity usually conducted by the vendor. These are only examples that are intended to illustrate the different approaches and their advantages and disadvantages, while the advantages for each approach can be considered disadvantages of the other. Within this context, every organization that wants to introduce a new IS has to decide which paradigm – custom build or standardized – it wants to follow. The advantages and disadvantages each paradigm has influence this conflicting decision in one direction or the other.

However, there is a third alternative, which is closely analyzed throughout the remainder of this paper – CBSD. It is argued that this approach offers the opportunity to surmount the problem that advantages of one concept are automatically disadvantages of the other. Rather, the new approach combines the advantages of the two preceding paradigms and creates an opportunity for a new, better way of software development – a truly dialectical change.

3. A Brief Background on Dialectical Change

Dialectical theory assumes that an entity that is subject to change exists in a complex environment with different conflicting and opposing forces that each try to pull the entity into a specific direction [41]. Within this environment stability is achieved by a balance of power between these different forces.

This concept of dialectical change is widely applied to explain change within organizations [6]. Considering the context of software development, the entity that is changed is an industry rather than a single organization. The generality of the concept of dialectical change, however, allows for such a different scope [41]. In the industry of software development, the above described advantages and disadvantages of standardized vs. custom made software each influence the decision of a potential buyer towards one solution or the other. The today existing balance of standardized and custom made software – which is obviously highly skewed towards the standardized solution – can be considered the outcome of this struggle.

Under certain circumstances, dialectical theory assumes that one paradigm is not simply superseded by another, but that both paradigms engage in conflict. Through this conflict, the old paradigm is replaced – but not entirely – by the new one; rather the parts of the old paradigm that proved to be inferior to the corresponding parts of the new paradigm are replaced. Consequently, those parts that proved to be superior to the corresponding parts of the new paradigm are retained. The emerging paradigm is thus neither the old paradigm, nor the new paradigm, but a synthesis of the two. This synthesis is supposed to be superior to both preceding paradigms [41].

However, as mentioned above, this emergence of a new, better paradigm is by no means guaranteed. Van de Ven & Poole [41] argue that if one paradigm mobilizes enough forces, it may simply overpower the other paradigm. This seems to have happened throughout recent years, when standardized software replaced custom made solutions to a significant degree. Nevertheless, dialectical theory offers valuable insight into the process of changing the software development industry. Benson [6] mentions four aspects of dialectical change that are of prime

relevance. First, dialectical change is rooted firmly in the *context* of change itself. Therefore, the change process has to be regarded from a perspective that takes into account the goals and interests of different existing stakeholders and their power to defend and enforce their interests. Not only software developers are important, but also other aspects – such as the customers of the software industry or the technical environment.

Second, Benson [6] describes dialectical change as *total*. Because change results from existing structures, these structures have to be considered when analyzing change. Structures that emerged through prior change processes are the result of structures that lay even further in the past. Because of this, new emerging structures are intertwined with past structures through a multitude of complex connections. These connections are not always coherent, and are by no means completely understood. Therefore, dialectical change analysis, although focused on unique autonomous phenomena, should never loose sight of the complex whole of the environment. This perspective is of prime importance later in the paper, when the impact of traditional software development organizations on the change process is discussed.

The third aspect of dialectical change, as described by Benson [6], is *contradiction*. The creation of new ideas as a basis of change results in the upcoming of inconsistencies, ruptures, and incompatibilities with traditional concepts – in short – contradicting ideas. These contradictions form the basis of dialectical change. New concepts arise that challenge old ideas. They might have emerged from partially autonomous organizations as responses to a changed environment, or as new ideas that appear in society. According to Benson [6], these contradictions perform three important – even essential – tasks in the context of dialectical change. They expose dislocations or crises that activate the search for alternative solutions, they facilitate or prevent change, and finally, they define limits of change for a given system.

The final aspect of dialectical change is *praxis* [6]. The fact that dialectical analysis itself facilitates change is subsumed under this point. By analyzing and objectifying current structures, individuals or organizations realize that they are not imprisoned in the prevailing structures. This empowerment of individuals or organizations to become an active agent of restructuring their environment is the final important aspect of dialectical change. Seen from this perspective, even this research contributes its share to the process of dialectical change in the software industry.

After this discussion on dialectical change and its different aspects, the paper continues with presenting the proposed synthesis in the software development industry – CBSD. Based on the above conducted discussion of standardized and custom made software, an environment is presented that obeys the assumption of combining the best traits of both of the preceding paradigms.

4. The CBSD ecosystem

Generally, proponents of CBSD describe a multi-tiered architecture of stakeholders that closely follows that described by Vitharana [42]: *component developers* are those parties that supply the marketplace with components (e.g. individual developers, IS departments, developing organizations, open source projects). Their main tasks are the designing, coding and unit testing of software components. *Application assemblers* utilize these publicly accessible components to assemble them into holistic systems. Consequently, their main tasks involve integrating, linking and testing the assembled systems. These systems are then adopted and applied by *customers*. It is important to note that the roles within this framework are by no means static. Rather, it is a highly dynamic concept in which stakeholders might act in two or even all three of the proposed roles, and, in which roles of stakeholders might change over time, even from project to project.

One reasonable enhancement of this model is the distinction between micro- and macro-components. While the first type of component offers a clearly defined, simple functionality, the second is already a complex arrangement of several micro-components and offers a higher degree of integration – yet with the disadvantage of narrower scope. Again, the distinction between micro- and macro-component is not static. What one stakeholder considers a micro-component may be a macro-component for a different stakeholder. This discussion results in a CBSD *ecosystem* including the different stakeholders mentioned above. A graphical illustration of this ecosystem is given in Figure 1. Please note that this is only a snapshot of the ecosystem at any given point in time. As indicated the concept is dynamic; also the choice of two levels of component developers is arbitrary, there might be more or less levels of developers.

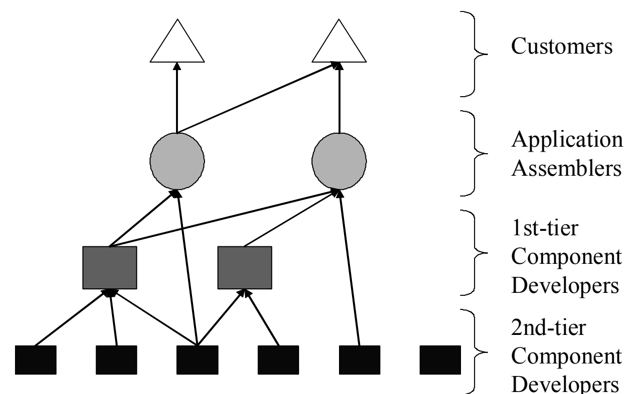

Figure 1. The CBSD Ecosystem

Within this framework, it quickly becomes obvious how the advantages of each preceding paradigm are combined by CBSD. The components can and should be standardized, off-the-shelf pieces of software (usually referred to as COTS – components off the shelf), with all the associated advantages. The combination of these COTS can be described as an individually designed IS – again with the advantages that come with this approach. When identifying the advantages of CBSD, the literature usually points towards these:

• *Reduced lead time*. The fact that systems are designed from already existing parts eliminates the time needed to develop these basic components. Consequently, the time needed to develop the whole system is reduced by a substantial amount [2, 25, 35, 42, 43].

• *Leveraged costs developing individual components*. The fact that components can and will be used in several systems results in a higher demand for the developed piece of software. This obviously reduces the costs for each individual application [25, 35, 42, 43].

• *Enhanced quality*. The fact that components are produced by organizations that specialize on certain types of components leads to a steeper learning curve, and consequently higher quality of the components. Furthermore, each component is used in a number of different systems. This leads to a better quality control, in which errors are discovered faster and can be eliminated easier [2, 25, 32, 35, 42].

• *Maintenance of component-based applications*. It is argued that systems that are assembled out of several components can be easier maintained. The main reason for this is the easy replacability of components in a system [35, 42].

In order to achieve these advantages, there are two different aspects in the new paradigm that have to be emphasized. First of all, the monolithic IS that have been used so far – no matter whether they have been individually designed or bought off-the-shelf – need to be broken up into smaller units, i.e. components. Furthermore, once these components have been identified and created, they have to be deployed in multiple contexts. Only if these two principles are fulfilled, the here proposed structure can be successfully positioned against the two preceding paradigms. This can also be seen in definitions of what components actually are.[1] Messerschmitt & Szyperski [27] for example define a component with the following five characteristics:

• *Composable*. A component has to be able to be composed with other components. It is one of the core principles of CBSD to increase development productivity achieved through assembly of components.

• *Encapsulated*.For each component, only the interfaces are visible and the implementation cannot be modified. This avoids redundant variations; furthermore, all uses of a component benefit from a common maintenance and upgrade effort [30].

• *Unit of independent deployment and versioning*. A component can be deployed and installed as an independent atomic unit and later upgraded independently of the remainder of the system. This allows customers to assemble and to *mix and match* components even during the operational phase, thus moving competition from the system to the component level.

• *Multiple-use*. A component has to be able to be used in multiple projects. This indicates that the development costs of each component can be shared over multiple uses.

• *Non-context-specific*. A component has to be designed independently of any specific project and system context. By removing the dependence on system context, components are more likely to be general and broadly usable.

When closer analyzing this definition, the two aspects mentioned earlier become more prevalent. The first three characteristics clearly point towards the concept of modularity as an underlying principle; the other two hint upon software reuse as underlying principle. Consequently, throughout the following two chapters, these two concepts are discussed from an explicit component viewpoint. They are then used to extract influencing factors on the dialectical change process that currently takes place in the software development industry.

5. Modularity

In his seminal paper *The Architecture of Complexity*, H. A. Simon [36] argued that complex systems – those that consist of a large number of parts and interact in a non-simple way – are often characterized by a *hierarchical setup*. In his perception, a hierarchical setup is one in which a system is composed of a set of interrelated subsystems; each of these subsystems is, in turn, composed of other subsystems, and so on, until finally a lowest level of elementary subsystem is reached. Obviously, these elementary subsystems are somewhat arbitrarily chosen. However, such a distinction has to be made at some point.

Without a doubt, the here discussed CBSD ecosystem can be characterized as a complex system [10] in the sense of Simon, and also obviously this system is

[1] There are considerable difficulties to reach a consensus for a definition of components. Most definitions are normative in nature by listing a range of characteristics that have to be fulfilled by a piece of software to be a component. A simple, yet comprehensive positive definition is not known to the authors.

characterized by a rather high degree of hierarchy (refer to Figure 1 for a graphical representation of this hierarchy).

Referring to Simon [36] again, it can also be argued that the CBSD ecosystem follows the second feature discussed in his article – *near decomposability*. This concept describes the fact that in a system the interactions within the subsystems are of magnitudes more intense than those interactions among different subsystems. This near decomposability of systems – referred to at other occasion as *loose coupling* [29] – is a concept that deserves closer attention. In their discussion on loosely coupled systems, Orton & Weick [29] propagate their perception that there are two different interpretations of loose coupling. One is the unidimensional interpretation – when following it, loose coupling is seen as an extreme on the continuum between loose and tight coupling. The more intense the interconnections are, the tighter a system is coupled. Contrary to this, the dialectical interpretation sees loose coupling as the coexistence of connection and autonomy. In this interpretation, parts of loosely coupled systems strive for autonomy, while at the same time retaining connections with the other parts. Obviously this perception is more complex than the unidimensional one; as Orton & Weick [29] state it: "dialectical concepts are rare, because they are difficult to build" (p. 216).

Nevertheless, this interpretation of loose coupling is chosen to guide the further proceeding of this research. An illustrative example how to utilize the concept of loose coupling in an organizational setting is given by Dubois & Gadde [17]. They argue that the construction industry can be characterized as a loosely coupled system, because construction companies in general experience a high degree of autonomy but enter tight connections with each other, when conjointly working on a project. These ideas can be transferred to the CBSD ecosystem, where *component developers* are highly autonomous unless an *application assembler* brings them together for a collaborative development project. Following Dubois & Gadde [17], Figure 1 can be adapted to include these ideas. Figure 2 represents the changed and simplified structure.

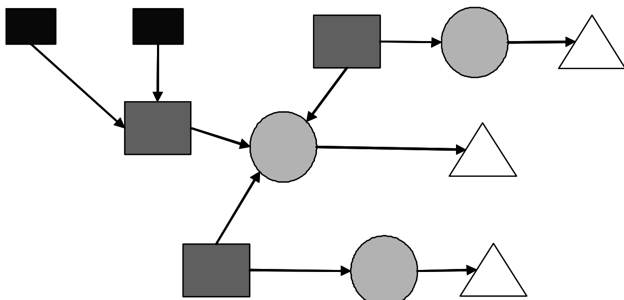

Figure 2. CBSD Projects as Loosely Coupled Systems. Source: Adapted from Dubois & Gadde [17].

Within the here described situation – which is just like Figure 1 again only a snapshot at a certain time – components of all three 1st tier component developers are used within one project, and are thus dependent upon each other, which is an indication for a strong connection. On the other hand the two 2nd tier component developers are only delivering to one 1st tier developer, with which they are highly interconnected. Contrary to this, they are highly autonomous with consideration of the other 1st tier developers. The fact that the different actors in this system strive for autonomy, while still having close interactions when collaborating in a joint project constitutes the situation of a loosely coupled system in the dialectical sense of the concept [29]. Within this context, however, the underlying assumptions for CBSD always have to be considered. Especially the *non-context-specificity* and the fact that components are *units of independent deployment and versioning* are important. Thus, the coupling is not tight in a sense that the components have to be adapted to the context of the project, but rather in a sense that a component-dependent training or testing is necessary. This seems to be inevitable, because today's components unfortunately are not as far developed as other technologies – like computer hardware – and still do not offer sophisticated plug-and-play technologies [8].

After this excursion on the basics of loosely coupled systems and modularity, these concepts are now applied in the context of dialectical change of the software industry towards CBSD. An interesting point in this context is raised by Schilling [34] who analyzes environmental factors that explain a shift in product development towards a more or less modular design. The model explicitly draws upon heterogeneity of inputs and demands, which are both indicators for a highly complex environment, as the two driving forces that increase modularity in product design. These factors are furthermore mediated by the construct of urgency. This is seen as a collection of mediating factors that either increase or decrease the pace of the transition towards a more or less modular design. For example, the speed of technological change or the competitive intensity for a product – again two indicators for a highly complex environment – can be seen as increasing the pace towards a more modular design.

This perception is in line with that of other authors that discuss modular product design. Baldwin & Clark [3], for example, offer the following three advantages of modular designs, which closely reflect the perception of Schilling that modularity is most appropriate in an uncertain, complex environment:

• *Modularity increases the manageable complexity*: Complex systems easily surpass the cognitive abilities of any single individual. Therefore breaking the system into modules allows a studying of one module at a time.

• *Modularity enables the parallelization of processes*: The information hiding between modules allows for a development of a module without internal understanding of the other modules.

• *Modularity reduces uncertainty*: Necessary future system changes that result from a changed environment can be better enforced. They do not affect the entire system, but are confined to the relevant modules, which are relatively independent from each other.

These factors are also mirrored in literature from the IS community. In his groundbreaking paper *On the Criteria to be used in Decomposing Systems into Modules,* D. L. Parnas [30] offers almost identical benefits when discussing the effects of modular programming. Furthermore, Fan, Stallaert, & Whinston [18] argue that component based systems are especially appropriate for highly dynamic environments with fast-paced changes. Reconsidering the above given advantages of CBSD (e.g. reduced lead-time, increased flexibility), this comes as no surprise – flexible systems offer a good *fit* with complex environments, and thus promise more success [28].

There exists a multitude of examples for this development. One commonly cited example is the automotive industry in the early 20th century. An increasing complexity of the products forced car manufactures switch to modular product design. It is even argued by Womack, Jones & Roos [46] in their highly influencial book *The Machine that Changed the World* that this development has to be seen as the key factor for mass production in the automotive industry: "The key to mass production wasn't (…) the assembly line. Rather, it was the complete and consistent interchangeability of parts and the simplicity of attaching them to each other" (p. 26 ff).

Another case in point for this kind of development is offered by Baldwin & Clark [3] and their discussion of the first modular design of a computer. Reacting on the pressures of its customers and the U.S. Justice Department in an increasingly complex environment for computer hardware, IBM decided to implement a modular product design. The adoption of modular design by IBM is not merely another example for the advantages of exactly this design. It also offers another important perspective. Sanchez & Mahoney [33] argue for a close connection between modular product design and modular organizational design. While tightly coupled products require a "tightly coupled organization structure coordinated by managerial authority hierarchy, an organizational design typically achieved within a single firm" (p. 65), loosely coupled products can be developed within an organizational system that is less integrated, up to a point where organizations that conjointly work on a product are not even part of the same company. This is exactly what has happened to IBM and the computer hardware industry. Before the decision towards modularity in the 1960s, there was an absolute dominance of IBM in the computing machinery market. After the decision the market structure changed – albeit slowly – towards a more diversified setup.

Figure 3 visualizes this dominance through the topmost, bold curve that represents the market capitalization of IBM and the other, shaded curves that represent the market capitalization of other sectors of the computer industry.

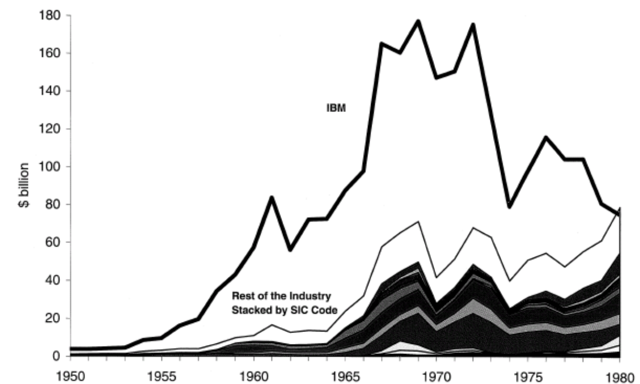

Figure 3. Market Capitalization in the Computer Industry. Source: Baldwin & Clark [3].

Summing up the discussion of modularity in the software development industry, it can be stated that the complex IS environment clearly favors a modular product design. This modular product design in turn should lead to modular organizational structure – exactly what has been described as CBSD *ecosystem*.

6. Reuse

As already indicated, the concept of software reuse is the second building block of CBSD. It is analyzed in more detail in this section – albeit always from the perspective of the application assembler in the CDSD ecosystem.

Software reuse is a quantitatively very well researched field. This research indicates that there is evidence that software reuse can indeed contribute its share to the goals of reducing lead time, leveraging costs, and enhancing quality in software development mentioned for the CBSD paradigm. Reports of cost savings in the millions of dollars are not uncommon [31]. Albeit the inherent weakness of the measurement of software productivity through lines of code (LOC) several studies use this measure to show the effects of software reuse. One such

research effort about increased productivity reports an increase from 500 to 800 LOC without reuse to 800 to 3200 LOC with reuse [26]. Furthermore, there are reports of reducing cycle time of software projects by up to 44 percent through reuse of software [23]. There are also studies that report a significant improvement in the number of defects per LOC ratio [13]. Finally, two case studies that summarize and offer a perspective from practice on the topic of reuse of software can be found in Apte et al. [1] and Banker & Kauffman [4].

While these findings seem to emphasize the importance the concept of software reuse has on the here proposed framework of CBSD, it has to be mentioned that there are differences between the two approaches. The probably most striking difference is the fact that software reuse is just that – the reuse of already existing software to increase productivity for newly developed software.

Contrary to that, CBSD is characterized by the concepts of *multiple-use* and *non-context-specificity*. This indicates that components are produced independently of any future use and are deployed across organizational boundaries – ideally over some type of market place. While this seems intuitively clear from Figure 1, it is not considered in any of the above cited works on software reuse. This concept is actually so contrary to the perception of the mainstream reuse literature, that Barnes & Bollinger [5] argue that "most reuse producers (vendors) and reuse consumers (buyers) differ from their commercial counterparts primarily in that the transfer of products between them takes place within a single company or project, rather than across company or organizational boundaries" (p. 15).

Obviously, this fact adds to the complexity of the CBSD ecosystem by introducing transaction costs into the framework. The first definition of transaction costs is usually accredited to R. H. Coase [12] and his work on *The Nature of the Firm*. He defines transaction costs in the broadest sense as "costs of using the price mechanism"[2] (p. 390). Consequently, these costs can be saved through the integration of certain activities within one organization with a central authority. Following Coase [12], it is exactly this trade-off between costs that accrue when using a central authority and costs that arise when using the price mechanism that determines the size of a firm: "A firm will tend to expand until the costs of organising an extra transaction within the firm become equal to the costs of carrying out the same transaction by means of an exchange on the open market" (p. 395).

Building on this theory, Williamson [45] introduces the concept of the *efficient boundary*. In this concept, it is assumed that a product of any kind consist of several components that can either be bought on the open market or can be produced in-house. Consequently, the efficient boundary of an organization is the boundary which includes all the activities that an organization decides to conduct in-house. The main influencing factor of this decision is *asset specificity*, which is defined as "the degree to which durable, transaction-specific investments are required to realize least cost supply" [45] (p. 555). The main emphasis of this definition is clearly on the *transaction-specific* nature of the investments. Asset specificity only becomes an issue, when changing from one transaction partner to another has severe financial consequences.

Now mapping asset specificity to the efficient boundary of an organization, it is argued that for assets which are of low specificity, markets generally enjoy cost advantages; while for high asset specificity, integration within an organization is more efficient: "classical market contracting will be efficacious whenever assets are nonspecific to the trading parties; bilateral or obligational market contracting will appear as assets become semispecific; and internal organization will displace markets as assets take on a highly specific character" [45] (p. 559). Thus, with increased asset specificity, intra-organizational production is seen as superior to the open market. Furthermore, it has to be stated that one of the key complementary factors to asset specificity is uncertainty. With greater uncertainty the difficulty to manage the interface of a boundary-spanning transaction increases even more. This argument is also reflected in the literature on IS. Schilling [34] explicitly mentions the uncertain and highly specific nature of software products as a reason for their seamless integration into *suits* as an example.

Thus, the economies of software reuse can partially be offset by transaction costs that arise for acquiring reusable software in the market. The costs of acquisition are particularly high in uncertain environments and when software products are very specific. Moreover, economies of reuse also diminish when software products are highly specific.

Consequently, a higher degree of uncertainty and a higher degree of specificity of the product, should favor a higher degree of integration within one organization. Obviously, this is contradictory to the findings of the analysis of the concept of modularity, which stated that a higher degree complexity should lead to higher degree of fragmentation in a system. Thus, the subsequent section is concerned with an explanation of this tension on the basis of the theory of dialectical change discussed above.

[2] Usually *transaction costs* are defined as those costs that are not contained in the production process. Thus, these costs usually are split into the five phases of *initiation*, *agreement*, *control*, *adaptation*, and *cancellation* of a contract [15]. From this perspective, it might be argued that the costs for the adaptation of the software that are necessary in the context of CBSD are not *transaction costs*. However, using the broader definition given above, these costs are also included in the *transaction costs* for this study.

Following this, the concept of *mindfulness* is introduced as a proposed way to resolve the contradiction.

7. Dialectical Change as an Explanation

When discussing the theory of dialectical change above, it has been argued that the change cannot be considered without also considering all the stakeholders with their goals and interests, the existing structures which lead to the change, the contradictions with traditional concepts, and the fact that change has to be rooted in *praxis*. When considering these underlying principles, one can gain a better appreciation for the state of today's software development industry.

Thinking about stakeholders of the industry and their goals and interests, it is obvious that today the industry is dominated by a rather small number of relatively large software producers. These organizations do not have any intention to abandon their dominant position in the industry. This has also been noted by Schilling [34]: "Firms might wish to prevent the adoption of modular product designs because modularity would decrease their market power or architectural control" (p. 326). This dominance in the software industry leads for example authors like Dreiling et al. [16] to statements like the following: "The developments of the past five years thus give ample indication that integration in enterprise systems appears to be only viable by accommodating with the circumstance that systems development remains in the hands of a few 'global players'."

However, while realizing that truly modular software design would harm the dominant role of these large corporations in the industry, they also consider the example of IBM and the computer hardware market. As mentioned above, IBM has been forced to adopt a modular product design, but failed to form the emerging modular hardware market according to its believes. Greenbaum [21] argues that large software producers already have embraced the CBSD paradigm. However, they are by no means willing to foster the emergence of an ecosystem in the sense of this paper. Rather they struggle for an adoption of CBSD processes internally, creating an ecosystem within their existing organization. Through this, they want to impose their technologies and methods onto this new paradigm through tightly interconnecting the new modular system with their old, monolithic systems. This proceeding is contrary to what IBM did both with the System/360 and the personal computer as well [14], and allows the larger software companies to exploit the advantages of the proposed paradigm without having to sacrifice their dominant position in the industry.

Furthermore, this approach also covers the need to address contradictions with traditional approaches as well as the need to root a dialectical change in praxis. By interconnecting the newly developed modular systems with the old, monolithic systems, organizations that invested no small amount of time and money [7] into the development of these systems can adopt the new, modular systems without forsaking these investments. Furthermore, issues around practical aspects like necessary developer know-how can possibly be mitigated through this approach.

8. Outlook and a Possible Solution

Within this context, it has to be noted that despite the fact that creating an ecosystem within an organization might yield some of the benefits of the proposed ecosystem, the full range of advantaged can only be achieved through the separation of tasks across organizational boundaries. While *reduced lead time* and *ease of maintainability* might be achievable within an organization, *leveraged costs* and *enhanced quality* are benefits that can be better realized in the increased competition of an open market like the ecosystem.

Consequently, the only question that remains to be answered is how such an ecosystem can be fostered. A possible solution to this problem could be the concept of *mindfulness* recently introduced in the IS literature [39]. Stemming from an individual level in the cognitive science, mindfulness is defined as having five different aspects: Openness to novelty, alertness to distinction, sensitivity to different contexts, awareness of multiple perspectives, and orientation in the present [38]. Transferring this concept to organizations [44] it has been argued that mindful organizations are especially successful in making effective decisions when it comes to managerial fashions and fads [19].

This is an ability that can be considered highly important for managers of small software development organizations that try to find their niche in the CBSD ecosystem. Obviously, these organizations need to have a high degree of openness to novelty to accept the new paradigm here proposed. Also, they need alertness to distinction, sensitivity to different contexts, and awareness of multiple perspectives to identify possible component based solutions that yield a viable business-model and to successfully manage the deployment of their component(s) in different contexts. Furthermore they need an orientation in the present, to overcome the obstacles that the current structure of the software development industry poses for the proposed paradigm shift.

9. Summary and Conclusion

The research effort presented here starts from the early beginnings of software development, when custom developed software has been the rule. Continuing with the emergence of standardized software packages, a

dialectical change perspective is utilized to argue for a new paradigm of software development: CBSD. This new paradigm is then analyzed in depth through the perspectives of software reuse and software modularity, which both form essential parts of the approach. Each of these leads to conflicting conclusions about the adoption of the new paradigm. Again referring to the theory of dialectical change, the current state of the software development industry is discussed with special consideration of the different stakeholders and their goals and interests. Arguing on this basis, the CBSD ecosystem cannot emerge as a dominant paradigm for software development. However, small niches might offer considerable freedom for smaller software developers to apply the new approach. Finally, the concept of mindfulness is introduced to give an idea how these small software developers can occupy their niche.

Obviously, this paper is theoretic in nature. Consequently, it can only be seen as a first beginning on a long way of understanding the process of change that the software industry is currently undergoing. Nevertheless, the fact that software developers are continuously searching for ways and means to increase their productivity strongly points towards an adoption of the component based paradigm. A closer analysis of this process of adoption, especially under consideration of the foundation laid in this paper, is the logical next step to gain a deeper understanding of the future of software development.

10. Acknowledgments

This work is a result of the project CollaBaWue which researches collaborative software development and is supported by the federal state of Baden-Wuerttemberg. The project is part of the research association PRIMIUM.

11. Literature

[1] Apte, U., Sankar, C. S., Thakur, M., & Turner, J. E., “Reusability-Based Strategy for Development of Information Systems: Implementation Experience of a Bank”, *MIS Quarterly*, Vol. 14, No. 4, pp. 421 – 433, 1990.

[2] Baik, J., Eickelmann, A., & Abts, C., “Empirical Software Simulation for COTS Glue Code Development and Integration”, *Proceedings of the 25th Annual International Computer Software and Applications Conference*, Chicago, USA, 2001.

[3] Baldwin, C. Y. & Clark, K. B., *Design Rules – Volume 1. The Power of Modularity*, The MIT Press, Cambridge, USA, 2000.

[4] Banker, R. D. & Kauffman, R. J., “Reuse and Productivity in Integrated Computer-Aided Software Engineering: An Empirical Study”, *MIS Quarterly*, Vol. 15 No. 3, pp. 375 – 402, 1991.

[5] Barnes, B. H. & Bollinger, T. B., “Making Reuse Cost-Effective”, *IEEE Software*, Vol. 8, No. 1, pp. 13 – 24, 1991.

[6] Benson, J. K., “Organizations: A Dialectical View”, *Administrative Science Quarterly*, Vol. 22, No. 1, pp. 1 – 22, 1977.

[7] Bingi, P., Sharma, M. K., & Godla, J. K., “Critical Issues Affecting an ERP Implementation”, *Information Systems Management*, Vol. 16, No. 3, pp. 7 – 14, 1999.

[8] Bohem, B. & Abts, C., “COTS Integration: Plug and Pray?”, *IEEE Computer*, Vol. 32, No. 1, pp. 135 – 138, 1999.

[9] Brehm, L., Heinzl, A., & Markus, M. L., “Tailoring ERP Systems: A Spectrum of Choices and their Implications”, *Proceedings of the 34th Hawaii International Conference on Systems Science*, Hawaii, USA, 2000.

[10] Brooks, F. P., *The Mythical Man-Month*, Addison-Wesley, Boston, USA, 1995.

[11] Campbell-Kelly, M., “Development and Structure of the International Software Industry, 1950 – 1990”, *Business and Economic History*, Vol. 24, No. 2, pp. 73 – 110, 1995.

[12] Coase, R. H., “The Nature of the Firm”, *Economica*, Vol. 4, No. 16, pp. 386 – 405, 1937.

[13] Coomer Jr., T. N., Comer, J. R., & Rodjak, D. J., “Developing Reusable Software for Military Systems – Why it is Needed and Why it isn’t Working”, *ACM SIGSOFT Software Engineering Notes*, Vol. 15, No. 3, pp. 33 – 38, 1990.

[14] Costello, M. C. & Gomes-Casseres, B., “The Global Computer Industry”, *Harvard Business School Case*, Ref.# 9-792-072, 1992.

[15] Dibbern, J., *The Sourcing of Application Software Services*, Springer, Heidelberg, GER, 2004.

[16] Dreiling, A., Klaus, H., Rosemann, M., & Wyssusek, B., „Open Source Enterprise Systems: Towards a Viable Alternative“, *Proceedings of the 38th Hawaii International Conference on Systems Science*, Hawaii, USA, 2005.

[17] Dubois, A. & Gadde, L. E., “The construction Industry as a Loosely Coupled System: Implications for Productivity and Innovation”, *Construction Management and Economics*, Vol. 20, No. 7, pp. 621 – 631, 2002.

[18] Fan, M., Stallaert, J., & Whinston, A. B., “The Adoption and Design Methodologies of Component-Based Enterprise Systems”, *European Journal of Information Systems*, Vol. 9, No. 1, pp. 25 – 35, 2000.

[19] Fiol, C. M. & O'Connor, E. J., "Waking Up! Mindfulness in the Face of Bandwagons", *Academy of Management Review*, Vol. 28, No. 1, pp. 54 – 70, 2003.

[20] Fuggetta, A., "Open Source Software – An Evaluation", *The Journal of Systems and Software*, Vol. 66, No. 1, pp. 77 – 90, 2003.

[21] Greenbaum, J., "Build vs. Buy in the 21st Century", *Intelligent Enterprise*, Vol. 6, No. 7, pp. 26 – 31, 2003.

[22] Haigh, T., "Software in the 1960s as Concept, Service, and Product", *IEEE Annals of the History of Computing*, Vol. 24, No. 1, pp. 5 – 13, 2002.

[23] Henry, E. & Faller, B., "Large-Scale Industrial Reuse to Reduce Cost and Cycle Time", *IEEE Software*, Vol. 12, No. 5, pp. 47 – 53, 1995.

[24] Kuhn, T., *The Structure of Scientific Revolutions*, 2nd ed., University of Chicago Press, Chicago, USA, 1970.

[25] Lindqvist, U. & Jonsson, E., "A Map of Security Risks Associated with Using COTS", *IEEE Computer*, Vol. 31, No. 6, pp. 60 – 66, 1998.

[26] Matsumoto, Y., "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels", in T. Biggerstaff (Ed.) *Software Reusability, Part 2: Applications and Experience*, Addison-Wesley, Reading, MA, 1989.

[27] Messerschmitt, D. G. & Szyperski, C., *Software Ecosystems: Understanding an Indispensable Technology and Industry*, MIT Press, Cambridge, USA, 2003.

[28] Miles, R. E. & Snow, C. C., "Fit, Failure, and the Hall of Fame", *California Management Review*, Vol. 26, No. 3, pp. 10 – 28, 1984.

[29] Orton, J. D. & Weick, K. E., "Loosely Coupled Systems: A Reconceptualization", *The Academy of Management Review*, Vol. 15, No. 2, pp. 203 – 223, 1990.

[30] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, Vol. 15, No. 12, pp. 1053 – 1058, 1972.

[31] Poulin, J. S., Caruso, J. M., & Hancock, D. R., "The Business Case for Software Reuse", *IBM Systems Journal*, Vol. 32, No. 4, pp. 567 – 594, 1993.

[32] Prahalad, C. K. & Hamel, G., "The Core Competence of the Corporation", *Harvard Business Review*, Vol. 68, No. 3, pp. 79 – 91, 1990.

[33] Sanchez, R. & Mahoney, J. T., "Modularity, Flexibility, and Knowledge Management in Product and Organizational Design", *Strategic Management Journal*, Vol. 17, Special Issue: Knowledge and the Firm, pp. 63 – 76, 1996.

[34] Schilling, M. A., "Toward a General Modular Systems Theory and its Application to Interfirm Product Modularity", *Academy of Management Review*, Vol. 25, No. 2, pp. 312 – 334, 2000.

[35] Sedigh-Ali, S., Ghafoor, A., & Paul, R. A., "Metrics and Models for Cost and Quality of Component-Based Software", *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Hokkaido, Japan, 2003.

[36] Simon, H. A., "The Architecture of Complexity", *Proceedings of the American Philosophical Society*, Vol. 106, No. 6, pp. 467 – 482, 1962.

[37] Stedman, C., "ERP Problems Put Breaks on Volkswagen Parts Shipment", *Computerworld*, Vol. 34, No. 1, p. 8, 2000.

[38] Sternberg, R. J., "Images of Mindfulness", *Journal of Social Issues*, Vol. 56, No. 1, pp. 11 – 26, 2000.

[39] Swanson, E. B. & Ramiller, N. C., "Innovating Mindfully with Technology", *MIS Quarterly*, Vol. 28, No. 4, pp. 553 – 582, 2004.

[40] Szyperski, C., Gruntz, D., & Murer, S., *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley, Reading, MA, 2002.

[41] van de Ven, A. H. & Poole, M. S., "Explaining Development and Change in Organizations", *Academy of Management Review*, Vol. 20, No. 3, pp. 510 – 540, 1995.

[42] Vitharana, P., "Risks and Challenges of Component-Based Software Development", *Communications of the ACM*, Vol. 46, No. 8, pp. 67 – 72, 2003.

[43] Voas, J. M., "Certifying Off-the-Shelf Software Components", *IEEE Computer*, Vol. 31, No. 6, pp. 53 – 59, 1998.

[44] Weick, K. E., Sutcliffe, K. M., & Obstfeld, D., „Organizing for High Reliability: Processes of Collective Mindfulness“, *Research in Organizational Behavior*, Vol. 21, pp. 81 – 123, 1999.

[45] Williamson, O.E., "The Economics of Organization: The Transaction Cost Approach," *American Journal of Sociology*, Vol. 87, No. 3, pp. 548 – 577, 1981.

[46] Womack, J. P., Jones, D. T., & Roos, D., *The Machine that changed the World*, HarperCollins Publishers, New York, NY, 1990.