World Academy of Science, Engineering and Technology
International Journal of Computer and Systems Engineering
Vol:3, No:8, 2009

Software Industrialization in Systems Integration

Matthias Minich, B. Harriehausen-Muehlbauer, and C. Wentzel

***Abstract*—**Today's economy is in a permanent change, causing merger and acquisitions and co operations between enterprises. As a consequence, process adaptations and realignments result in systems integration and software development projects. Processes and procedures to execute such projects are still reliant on craftsman-ship of highly skilled workers. A generally accepted, industrialized production, characterized by high efficiency and quality, seems inevitable.

In spite of this, current concepts of software industrialization are aimed at traditional software engineering and do not consider the characteristics of systems integration. The present work points out these particularities and discusses the applicability of existing industrial concepts in the systems integration domain. Consequently it defines further areas of research necessary to bring the field of systems integration closer to an industrialized production, allowing a higher efficiency, quality and return on investment.

***Keywords*—**Software Industrialization, Systems Integration, Software Product Lines, Component Based Development, Model Driven Development.

I. Industrialization

FROM a generic point of view, the term industrialization is defined as the dissemination of industries within an economy, in proportion to agriculture, handicraft and small trade. Relating to the production of goods and services, it is defined as the implementation of standardized and highly productive methods in order to increase efficiency and reduce cost [1, 2]. The process of industrialization began at the end of the 18th century in Great Britain and was characterized by an increasing division and specialization of labor, capital intensive technologies, mass production, rationalization and the application of new energy sources [2]. Industrialization is seen as a necessary step for economic growth, technological advances and increasing wealth. Only industrial production methods allow to produce a multiplicity of goods in a sufficient amount and quality [2].

The present paper will focus on the application of standardized and highly productive methods to the field of software development in systems integration. Applied to the process of industrialization as introduced above, the key concepts of such methods can be summarized in specialization, standardization & systematic reuse, and automation.

As of today, the above principles can be found in almost all industries at different levels of penetration. Standardization and specialization advance the level of reuse and enable automation of rote and menial tasks, whereas creative tasks (that cannot be standardized), such as product design, are still performed by highly skilled workers. Omitting the availability of the required commodities and energy, the fundamental principles of industrialization can be described as follows.

A. Specialization

In the given context, the term specialization describes the concentration of an economic subject (worker, business, society, etc.) to a particular area within a larger scope, such as certain industries, product families, technologies or skills. A production process is subdivided into less complex functions that can be assigned to well-trained workers or purpose-built machinery. This division of labor allows the specialization of individuals, expanding their knowledge and abilities in a particular area. In turn, they achieve a higher efficiency and quality. Specialization also allows reusing production or product artifacts. The former for example include processes, tools and machinery, while the latter include architectures, frameworks and components. Systematic reuse can only occur in a precisely delimited scope, defined by specialization and standardization [3].

The disadvantages of specialization lie in a reduced flexibility and thus the dependency on market demand of the area or skill in scope, as well as the dependency of upstream production. A highly specialized economic subject cannot quickly change its area of focus. A farsighted, strategic planning of specialization is mandatory.

Well known implementations of specialization can, for instance, be found in the automotive sector. A whole industry subcontracting to automotive manufacturers emerged, specializing in certain product families such as engines, brake systems or electronic control units. Furthermore, employees specialize in particular skills and tasks in the production process.

M. Minich is with the University of Plymouth, Plymouth, PL4 8AA United Kingdom and the University of Applied Sciences Darmstadt, 64283 Darmstadt, Germany (phone: +49 1713885673; e-mail: matthias.minich@plymouth.ac.uk).

B. Harriehausen-Muehlbauer, Prof. Dr., is with the University of Applied Sciences Darmstadt, 64283 Darmstadt, Germany (phone: +49 6151 16-8485; e-mail: b.harriehausen@fbi.h-da.de).

C. Wentzel, Prof. Dr., is with the University of Applied Sciences Darmstadt, 64283 Darmstadt, Germany (phone: +49 6151 16-8459; e-mail: c.wentzel@fbi.h-da.de).

B. Standardization & Systematic Reuse

Standardization describes the unification of specific attributes of production or product artifacts. The objective is to establish a common understanding of these attributes in order to exchange artifacts, integrate upstream work products, align production processes or simplify information exchange [4]. Together with specialization, standards provide the base for systematic reuse. Only if an artifact follows clearly defined principles, it can be reused as is in another product. Standards can be officially defined (by a binding regulation or contract), de facto (by market position or dominant usage), or voluntary.

With regard to market position or profit margin they can also be disadvantageous as standards encourage competition between suppliers. Furthermore, they may require tradeoffs in functionality that may affect a unique selling point of the own product or the lack of customization possibilities.

A good example of standardization can be found in the modular construction system of automotive manufacturers. Uniquely designed product artifacts such as axles or suspensions can be reused in many different models of a product family. Likewise, production artifacts such as assembly lines, tools or machinery, can be reutilized to produce many different products.

C. Automation

By division of labor, standardization, and systematic reuse, rote, menial or dangerous tasks can be taken over by purpose-built machinery. The operational sequence, regulation and monitoring of the production process is also performed by technical equipment. Such machines often are more precise and time & cost efficient as compared to human workers. Important prerequisites are specialization and standardization, as machinery cannot solve unknown problems. In an industrialized production, the worker's role shifts towards planning, monitoring and correction of the production process. The objective here also is to reduce cost and time and increase quality.

Drawbacks in automation inherit from the previously mentioned principles. High upfront investments require a minimum utilization rate to break even. Reduced flexibility implicates a high market dependency of the segment in scope.

Automation is as well an important factor in the automotive sector. The industry heavily relies on automated production such as welding robots or automated assembly lines.

II. Industrialized Software Development

Software development is "[…] slow and expensive, and yields products containing serious defects that cause problems of usability, reliability, performance and security" [3]. At the beginning of this chapter, industrialization was defined as a method to increase efficiency and quality and to reduce cost by implementing standardized and highly productive methods. The objectives of every software project can be categorized into quality, quantity, time and cost [5, 6]. Harry M. Sneed depicted their interaction as the Devil's Square [7], in which the four factors are in an antagonistic relationship. As the available productivity of the performing organization is limited and cannot satisfy all needs, tradeoffs have to be made. For example, doing more work in a higher quality will result in higher cost and a longer development time. However, by applying industrial methods and thus increasing productivity, quality and product complexity can possibly be increased and at the same time cost and production time reduced.

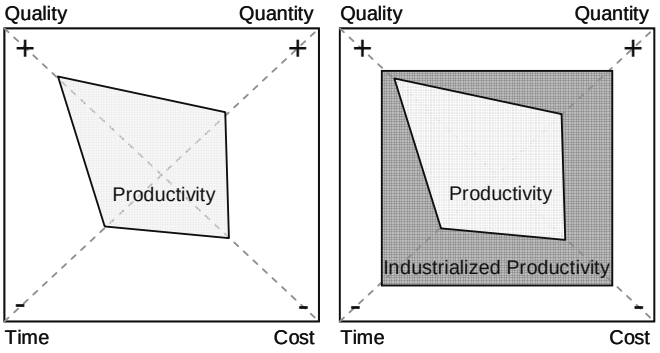

Fig. 1 Sneed's Devil's Square: Dimensional tradeoff versus industrialized productivity

Several efforts have been taken to apply such methods to software development. Referring back to the previous chapter, the key industrial principles now can also be found in the field of software engineering. *Specialization* is represented by Software Product Lines, *Standardization* & *systematic reuse* may be found in Component Based Development, and *Automation* can be achieved with Model Driven Engineering. Unfortunately, the most important concept, specialization, was invented last. As of gracious generality, caused by the lack of a clearly delimited scope, Component Based Development and Model Driven Engineering in their initial occurrence seem to have failed [3]. Only recently all the concepts are in place and can be used to facilitate industrialized software development, as for example described in Greenfield & Short's book "Software Factories" [3]. The referenced concepts will be briefly described in the following.

A. Software Product Lines

The latest and maybe most important concept is the one of Software Product Lines that maps to the industrial principle of specialization. It seems to be very difficult or even impossible to determine how mechanisms for reuse or automation should be implemented in an arbitrary context. Systematic reuse must be planned for and cannot occur coincidentally. A Software Product Line (SPL) therefore spans a clearly delimited frame around a family of software products, sharing "[…] a common, managed set of features satisfying the specific needs of a particular segment or mission" [8]. It first emerged in 1995 in a Swedish naval software firm and was further developed at the Carnegie Mellon Software Engineering Institute [8]. The concept requires to separate product development from product line development. The former produces the actual software product, while the latter produces

the required assets to support the development process. By concentrating on a clearly delimited scope, production assets can be much more powerful as, for instance, reusable components or frameworks and architectures. However, specialization alone would not allow for any diversification as required by different customers. Software Product Lines therefore identify recurring functionality and points of variation to define a common framework or architecture in which customer specific requirements can be considered. This concept is also known as mass customization in other industries.

During actual product development, knowledge and reusable assets, such as business functionality components, are captured to include them in the Software Product Line for future products. The following figure depicts the described concept:

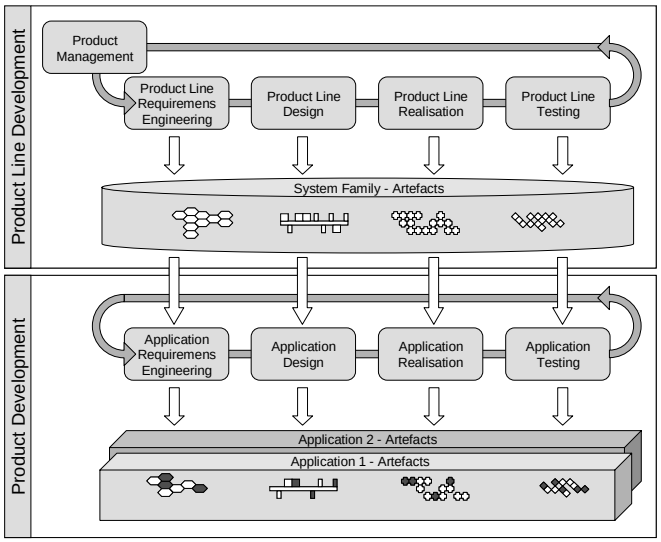

Fig. 2 Software development in a Software Product Line (q.v. [9])

Product line developers produce product and production assets, utilized by *product* developers to produce the particular family member. During product development new assets are created and fed back into the product line. It is therefore very important that any customer specific variability is developed with a potential reuse in mind. Further advantages can also be found in a higher quality and a shorter time to market: As reusable product assets are reviewed, implemented and tested in many different products, chances to find faults and correcting them are significantly higher [10]. Furthermore, a once identified fault can be corrected before it becomes evident in other products. Although time to market is higher in the beginning due to product line development, it decreases significantly once assets are in place that can be reused for each new product [10].

Of course, this specialization to a particular segment or mission is not for free. Upfront investments are required to define the scope and the initial asset base for the Software Product Line. Unlike in manufacturing industries, these costs cannot be recovered by economies of scale as software can be copied very easily and customer requirements are hardly the same. SPL must therefore focus on *economies of scope*, producing distinct but similar products, all based on a common set of functionality. Literature suggests about three systems to reach the break-even-point as compared to conventional, one-off development [8].

B. Component Based Development

One of the first ideas of using industrial principles came up in October 1986 on the NATO conference on software engineering. It can be mapped to the industrial principle of standardization which is the foundation for the exchange of artifacts and systematic reuse. In his contribution "Mass produced Software Components" [11], M.D. McIlroy suggested to develop applications by assembling previously produced components, as most of a software's functionality has already been developed or will be required in many other applications as well. In his book about component software [12], Szyperski defines such a component as follows:

"A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

As in manufacturing industries, systematic reuse requires a clearly delimited context. It is for example much easier to build GUI components for Microsoft's .NET platform than within an arbitrary context [3]. By using current component standards it is possible to encapsulate business logic within reusable software building blocks. The context in which this occurs can be set by Software Product Lines. They define a delimited scope to employ reusable components for the development of new product line members. Current CBD standards define the requirements such a component has to fulfill from a syntactic and semantic point of view [13]. They furthermore define interface specifications, component allocation and component interaction across different programming languages and platforms. The underlying architecture can also be provided by these technologies in a way that it becomes possible to completely implement the commonalities of a certain product line, while allowing to "plug in" customer specific requirements [3].

The four most widely adopted component standards today are Sun's Java Platform Enterprise Edition (Java EE), the Corba Component Model (CCM) by the Object Management Group, and Microsoft's Distributed Component Object Model (DCOM), as well as their .NET Framework. All of them support language independent integration of other components or systems by providing clearly defined interfaces and data types which can be accessed on a binary level. CCM and Java EE are also platform independent.

C. Model Driven Development

The final aspect of industrialization, automating certain tasks, can be achieved with Model Driven Development (MDD) and was initiated by Computer Aided Software Engineering (CASE) in the 1980s. It encouraged development

methods based on graphical representations of software (models) with state machines, structure diagrams or dataflow diagrams [14] to generate source code. The graphical representations, however, were too generic to precisely describe the intended solution and did poorly map to the underlying technologies. The result was highly complex source code which had to be altered by hand. The corresponding models were out of date very soon as the CASE tools could hardly depict manual changes to the code. Today, model driven engineering has been further advanced, overcoming the problems discovered with CASE tools.

Using visually represented models as a description of software, it aims to raise the level of abstraction in order to fill the gap between the problem solution and the technical implementation, bringing the latter closer to a vocabulary understood by subject matter experts [3]. Omitted details are subsequently added until executable software is available. Of course, this process is by far not trivial: The extensive degree of freedom and context sensitivity becomes an issue if the model is to be interpreted by a code generator. To overcome this issue, MDD combines Domain Specific Languages and Transformation Engines & Generators [14]. Both are uniquely designed for a particular application domain, reducing the degree of freedom and possible contexts by providing a clearly specialized vocabulary and grammar. Once a system has been defined with an appropriate DSL, the resulting set of models may be transformed into either intermediate models or directly generated into source code. Similar to component based software development, MDD requires a clearly defined context in which it occurs.

III. Systems Integration

Software systems are being developed and used for more than 40 years now and become more and more important in day to day business. At the same time, IT faces high demands in quickly adapting to new business requirements. As legacy systems often do not offer the flexibility to do so, new systems are implemented which need to interact with the existing IT landscape. It is often not possible to simply replace legacy applications due to the extreme cost involved. This situation inevitably leads to systems integration efforts, joining the different subsystems into a cohesive whole, in order to alleviate functionality or data access via a common interface [15, 16].

The term integration can either be defined as a *state* in which entities continue to exist after being integrated, or as the *process* of integrating them into a larger entity. *Integration as a state* defines classes by which the degree of integration of IT systems can be differentiated and evaluated. *Integration as a process* deals with the steps required to move an IT system from a given degree of integration to a higher one, which is done by merging distinct entities into a cohesive whole or integrating them into already existing systems [15, 17]. The present work follows the latter definition of the term integration, i.e. the process of integrating distinct entities into a cohesive whole.

This process of integration can be further divided into data integration and application integration [16, 18]. Data integration concentrates on the integration of different data sources, for instance, by data consolidation or data warehousing. Application integration in turn covers the combination of different software systems that support business processes. The integration of such systems is also referred to as Enterprise Application Integration (EAI) and depicts a core area in today's business engineering. However, several interpretations exist for the term [19]:

- EAI as an integration middleware solution
- EAI as a high level integration on a semantic or process level
- EAI as an integration framework architecture
- EAI as an approach to implement business requirements, including strategic and process related considerations, utilizing different integration techniques

The present work focuses on Enterprise Application Integration and adheres to the fourth definition of the term, i.e. EAI as an integration approach from a strategic, process and technology related perspective. It does so because each layer may have severe influence on its neighboring ones and thus may not be considered in isolation, as suggested by the first three interpretations.

A. Dimensions of Integration

Within the previously adopted definition of the term EAI, literature usually defines several layers or dimensions of integration. They start from a strategic and business process point of view, through process partitioning for different systems, to the actual data and functionality management on an implementation level.

In her book "Prozess- und Systemintegration", Vogler for example defines process, desktop and systems as the three sub domains of integration [19]. The process domain defines how business processes are depicted onto the IT landscape and how they support the overall workflow from a more strategic point of view. The second domain (desktop) defines when and how different (heterogeneous) applications are involved, and how they exchange information with the user (e.g., via a common user interface) or with each other. The underlying systems domain then defines which application accesses which data, how data exchange takes place, and how data redundancy is managed.

Hasselbring offers a similar classification in [20] by defining a business, application and technology architecture. He limits the term EAI to the second layer only, while applying interorganizational process engineering and middleware integration to the first and last layer, respectively. Despite the different interpretation of the EAI term, Hasselbring indeed considers the remaining aspects in his work.

A comparable classification can be found with Fischer in

[15], who identifies a business, organizational, functional and technical dimension. The business dimension defines which IT systems are required based on the strategic business needs. The organizational dimension aligns IT systems and workflows, and optionally adapts either. Data collection and storage of information (data integration), as well as controlling intermeshing activities (process integration) is done within the functional dimension. The fourth dimension (technology) aims at proper coupling of the different IT systems, independent of their location or underlying technology (systems interconnection).

Taking the previous definitions and explanations into consideration, the present paper defines the following three dimensions of integration:

1) Business Process

On the business process dimension the organizational objectives, structure and core business processes of an enterprise are characterized. They define which business functionality and information is required and how the involved IT systems must interact from a semantic point of view. Integration decisions on this dimension are usually driven by mergers & acquisitions, collaboration agreements, or realignment of company objectives.

2) Workflow

The workflow dimension subdivides a business process into distinct activities and maps these to the different IT systems. It defines the data sources and functionality required from the available IT systems from a technical point of view, as well as the interaction among each other and with the end users. On the business process dimension these data sources and functionalities map to the semantic steps of the business process. Integration decisions on this dimension may inherit from higher or lower dimensions, or are driven by process adaptations due to regulatory influences or improvement activities, for instance.

3) Technology

Information and communication infrastructure of an integrated systems landscape is implemented at the technology domain. It defines which applications may access which data or functionality, how this is done, and how data management (e.g., redundancy) takes place. Integration decisions on this dimension inherit from higher dimensions, or are driven by technological changes, such as introducing or replacing applications.

B. Common Problems in Systems Integration

Despite the fact that systems integration supports and simplifies the execution of business processes, it involves several particularities and challenges during implementation. Based on Vogler in [19], the following sections briefly describe the potential problems per integration domain:

1) Business Process Domain

New business processes are often defined from a semantic point of view or are based on preconceptions from earlier projects regarding their representation in IT systems. As the IT landscape has direct implications on the business processes, it is important to understand the integration relationships in order to *identify and choose an optimal solution*. In many companies, however, the particular situation is hardly known. Furthermore, business processes designers may not know about the solutions available on the market and often do not have the knowledge to design an overall concept.

Similarly, the unawareness of the particular IT landscape may lead to *unforeseen consequences*. Only minor changes in a process may lead to adaptations of the underlying systems, which in turn may require the adaptation of other processes due to changed interfaces or data structures. If these consequences were known early enough, business processes could be designed around them.

2) Workflow Domain

Subdividing a business process into workflows and depicting them ad hoc on different IT systems may lead to a *suboptimal degree of integration*. Due to time or cost constraints, these systems are often interconnected on a point to point basis instead of using shared integration architecture. In extreme circumstances this leads to n*(n-1) relationships, making later changes more and more complex.

In such an environment the *integration relationships may be unknown* due to insufficient documentation and the lack of a big picture. New implementations may be redundant and the consistency and integrity of interfaces cannot be ensured, which leads to unforeseen consequences for other systems. A prominent example was the Y2K problem where it was hardly known which systems rely on the data to be changed.

Enterprises still do not use a *methodological approach* with best practices or standardized processes for their systems integration projects. However, suitable methodology has been defined in literature during the last years but is not yet known or adopted in the industry. This also becomes evident as SI is not sufficiently considered in current software development models [21]. Integration projects are often done ad hoc and for a single purpose only that leads to the initially mentioned suboptimal degree of integration.

Due to uncoordinated efforts and the lack of methodologies, integrated systems show a *high complexity*, leading to increased time and cost for future adaptations.

Heterogeneity caused by the previous problems prevents the implementation of holistic *integration platforms or architectures* within enterprises. Although there are certain middleware systems or transaction monitors in place, these are usually not part of a bigger picture.

3) Technology Domain

From a technical point of view, *heterogeneity* is the major issue in systems integration. Depending on the differences, data representation and functionality, as well as underlying technologies must be aligned. The required effort thus disproportionally rises with the number of systems to be

integrated, unless a common architecture or platform is used.

Another big problem is the integration with *legacy applications*. These were often designed as stand alone solutions with no integration in mind. Obsolete data management, interfaces, or a lack of documentation or maintenance make integration extremely difficult. Furthermore these systems often cannot be altered or replaced and therefore impose restrictions on the overall integration concept.

The final issue lies in the *redundancy of data*. In integrated environments it becomes difficult to define which data resides where, how it is accessed and how redundancy is managed. Without such management, information may easily become outdated and inconsistent, leading to serious issues in business process execution.

IV. The Industrialization of Systems Integration

Today systems integration solutions are still implemented from scratch by utilizing traditional software development methods, such as the Waterfall Model or the V-Model. These however were designed with regard to monolithic systems, as integration was not of interest at the time of their development. Recent works such as the V-Model XT briefly reference integration with external environments [5] but still do not pursue a standardized and methodological approach. The result may be an "integrated monolithic system" with highly complex dependencies as described in section B above. Moreover, these development models do not incorporate the basic principles of industrialization and thus may not leverage potential improvements in cost, efficiency and quality as initially stated.

As discussed in chapter II, Software Product Lines, Component Based Development and Model Driven Engineering represent specialization, standardization & systematic reuse, and automation for software development. The respective concepts are well understood and first literature is available on combining them in factory like development environments, as for example in Greenfield and Short's book on Software Factories [3].

As shown in chapter III, SI comes with several particularities, distinguishing it from the domain of conventional software development. It has to challenge a multiplicity of technologies, inflexible legacy systems, once only technology combinations and a very high complexity. It seems disputable whether the concepts for industrialized software development in their original form can be applied to the field of systems integration, as depicted in the following:

A. Software Industrialization Concepts with Regard to SI Particularities

1) Software Product Lines

In Software Product Lines, design and development occur in a particular context, sharing common features and solving common problems. Product families may either be tailored around complete business solutions or a series of related products. They concentrate on reusable implementation artifacts, as well as frameworks, processes and tools.

With reference to systems integration, the multiplicity of different technologies, caused by high heterogeneity, inflexible legacy systems and different data sources, seems to be a major drawback to the definition of distinguished product lines. In a product line covering Customer Relationship Management (CRM) systems for example, products may be highly integrated with third party logistics and finance systems. Including support for any potentially attached systems undermines the advantages of a delimited context, while excluding them will force development to occur outside the industrialized concepts. An additional drawback is the de-facto development of one-off solutions per customer. Barely any solution operates in the same environment or is interconnected with the same type of systems. The initial set-up cost for software product lines may therefore be contraindicative as the return of investment cannot be ensured.

2) Component Based Development

According to Greenfield & Short [3], development by assembly with software components has certain requirements that must be met: Platform independent protocols (e.g., XML), self-description of components (formalized and enhanced meta-data within components), deferred encapsulation (allowing to interweave additional functionality), assembly by orchestration (machine controlled interaction and management of components), and architecture driven development (to promote the availability of well-matched components).

With regard to systems integration, the author does not see any major difficulties to technically apply development by assembly. However, the assembly approach relies on systematic reuse and thus on a methodical approach in a clearly delimited context that may not be easy to define as shown in 1). This context also has an influence on the availability of predefined software architectures, as well as the number of reusable components. Furthermore systems integration standards are not common as of today [19]. The most important challenge to be met is the definition of a component based systems integration architecture in which development by assembly may occur.

3) Model Driven Engineering

Model Driven Development, and in a greater sense Model Driven Engineering, raises the level of abstraction to reduce complexity and express business concepts more efficiently. It consists of domain specific modeling languages and model transformation engines & code generators. The former allow a context free description of the intended products of a product line, whereas the latter provide model transformation to a lower, more specific model or eventually the generation of source code.

For systems integration, the efforts required to define a domain specific language (DSL) could become an obstacle, especially if applied to product lines with a limited number of expected products. With reference to Software Product Lines,

the scope of a DSL cannot be clearly delimited as each product may need to be integrated with other external systems. Furthermore, to automate the development process by transforming models to a lower level or generating source code, transformation engines and code generators have to be implemented which also impose high set up cost.

B. Areas requiring further research

As can be seen in section A, existing concepts of software industrialization may not necessarily suit the particularities found in the field of systems integration. Thus further research is required to either adapt or enhance existing concepts, while considering how to align organizational structures to support the application of industrial production paradigms.

The focus of the present research is therefore aimed at the application of industrial production principles in the specific domain of systems integration from a solution provider's point of view. The research deals with the following areas.

1) Organizational Aspects

Organizational aspects focus on the surrounding conditions of industrialization in SI. They are reflected in roles, responsibilities, and corporate structures, and should be carefully considered before performing a paradigm shift throughout the organization. With specialization and SI particularities in mind, an organizational structure needs to be developed which enables systems integration providers to implement industrial concepts. This subsequently imposes the question whether enterprises can afford to organize themselves in fully featured Software Product Lines or if other forms of organization, for instance, shared service centers for product line definition and management, or a combination of both, are more feasible.

Therefore an organizational concept, describing the definition of divisions and departments of a systems integration provider, may shape up to be useful as foundation for industrialization.

2) Software Product Lines

Given a typical systems integration provider, an approach to implement software product lines in a way that they are neither too small nor too large, has to be developed. How can the wide variety of customer requirements, heterogeneity of integrated systems, and one-off developments be covered, without endangering the return on investment? As it delineates their scope, product line design for systems integration also has to bear the concepts of systematic reuse and automation in mind.

Further research must discuss the detailed requirements of Software Product Line implementation and identify ways of applying or adapting them in a systems integration context.

3) Component Based Development

Given that an expedient classification of software products into product lines or families has taken place, is it possible to define software components to be reused in different integration solutions for different customers? As shown before, component based development requires an adequate architecture in which it takes place. With reference to the common problems of systems integration, a combination of component based architectures and systems integration frameworks seems necessary.

In a joint analysis of existing component architectures and SI frameworks, it should be figure out if a combination of both is feasible and may be used as the technical foundation for software product lines.

4) Model Driven Engineering

The probably most ambitious objective of an industrialized software development is the automated creation of artifacts such as model transformations or code generation. For SI it offers interesting possibilities to resolve problems related to the business process and workflow domain of SI, such as integration consequences and depicting intersystem relationships. However, it is unclear to which degree an SI service provider can economically implement such a concept and if it can be used for different customers. The role and potential advantages of domain specific languages in the given context is also unknown. Are separate tools such as model transformators or code generators required for each product line or can their foundations be reused?

Based on the previous three aspects, the feasibility of Model Driven Engineering in systems integration should be analyzed and suggestions for the degree of its implementation derived. In this context MDE may shape up to be useful to solve SI related problems such as unknown integration consequences or intersystem relationships.

V. Conclusion & Research Approach

Systematic reuse of existing software artifacts hardly takes place and the majority of goods is still produced from scratch. With increasing complexity and size of today's IT systems, a generally accepted and industrialized production principle becomes necessary. Promising approaches, notably Software Product Lines, Component Based Development, and Model Driven Engineering, are currently being developed and implemented in practice, as described in chapter II.

However, as software engineering takes place in a wide variety of application domains, it cannot be assured whether the available industrialization models can be applied to every one of them. One of these domains is systems integration in which IT systems are adapted and interconnected to support new or changing business processes or requirements. To better understand the particularities of this field, chapter III depicts its substantial differences that are primarily the lack of knowledge about the integrated IT landscape of an enterprise, the lack of a methodological approach and integration framework, and a high heterogeneity of systems.

Chapter IV picks up these particularities and maps them to the introduced concepts of industrialized software engineering. The first section shows why these concepts cannot be applied to systems integration in their initial occurrence, while the second suggests further research to

advance the field of software engineering in systems integration towards an industrialized production process:

- Organizational aspects: Which changes are required to the organizational structure of a systems integration provider in order to implement industrial production methods in an economically feasible way?
- Software Product Lines: Is the concept of SPL in its original form viable for systems integration providers? How can the gap between a standardized product family and customer specific requirements in a highly heterogeneous environment be bridged at feasible cost?
- Component Based Development: CBD and SI require a specific architecture or framework. Can both be combined to form a basis on which Software Product Line development and systematic reuse can be built on? How can the high heterogeneity of systems to be integrated taken into account?
- Model Driven Engineering: To what extend does it economically make sense to implement MDE in systems integration? Does MDE offer additional benefits to SI as, for instance, an integration management approach?

The present work can be classified into the scientific area of business informatics as it covers matters from business (organizational forms of enterprises and product family management) and computer sciences (implementation of CBD and MDE). To meet concerns about the fact that very little of software engineering research finds its way into practice [22], research in the described area could be conducted in close collaboration with the industry. Thereby the approach of action research, aiming at the retrieval of scientifically proven procedures and guidelines and applying them in practice, seems to be suitable. The approach consists of three major phases: During the first phase, scientists and practitioners outline the problem definition and a first concept is developed, based on domain analysis and theoretical research. In the second phase the derived concepts are discussed with subject matter expert and subsequently implemented in practice. The third phase then reflects the results of the implemented solution and derives suggestions for improvement and further research, out of which a new cycle of action research can be initiated. For each of the above aspects at least one action research cycle will be accomplished, further ones may be added as needed.

The overall objective of the depicted areas of research may be a guideline which will draw a holistic picture of industrialized systems integration and provide a software development approach that addresses the application of industrial concepts in systems integration.

REFERENCES

[1] Butschek, F., Industrialisierung. 2007, Ulm: Ebner & Spiegel.
[2] Brockhaus-Enzyklopädie, in Brockhaus-Enzyklopädie. 2005, Brockhaus: Mannheim.
[3] Greenfield, J. and K. Short, Software Factories - Assembling Applications with Patterns, Models, Frameworks, and Tools. 1 ed. 2004, Indianapolis: John Wiley & Sons.
[4] Brockhaus-Enzyklopädie, in Brockhaus-Enzyklopädie. 2005, F.A. Brockhaus: Mannheim.
[5] Balzert, H., Lehrbuch der Software-Technik: Software Management. 2 ed. 2008, Heidelberg: Spektrum Verlag.
[6] Sneed, H.M., Software-Qualitätssicherung für kommerzielle Anwendungssysteme. 1 ed. 1983, Bergisch Gladbach: Verlagsgesellschaft Rudolf Müller.
[7] Sneed, H.M., Software Management. 1987, Cologne: Müller GmbH.
[8] Clements, P. and L. Northrop, Software Product Lines. 2007, Boston: Addison-Wesley.
[9] Linden, F.v.d., K. Schmid, and E. Rommes, Software Product Lines in Action. 2007, Berlin, Heidelberg, New York: Springer.
[10] Pohl, K., G. Böckle, and F. van der Linden, Software Product Line Engineering. 1 ed. 2005, Berlin, Heidelberg, New York: Springer.
[11] Software Engineering. in NATO Software Engineering Conference. 1968. Garmisch Partenkirchen: NATO Science Committee.
[12] Szyperski, C., Component Software: Beyond Object-Oriented Programming. 1998, Massachussetts: Addison-Wesley.
[13] Andresen, A., Komponentenbasierte Softwareentwicklung mit MDA, UML 2 und XML. 2 ed. 2004, Munich, Vienna: Carl Hanser Verlag.
[14] Schmidt, D.C., Model Driven Engineering. IEEE Computer, 2006. 39(2): p. 7.
[15] Fischer, J., Informationswirtschaft: Anwendungsmanagement. Lehr- und Handbücher zu Controlling, Informationsmanagement und Wirtschaftsinformatik 1999, Munich, Vienna: Oldenbourg.
[16] Leser, U. and F. Naumann, Informationsintegration. 2007, dpunkt: Heidelberg.
[17] Riehm, R., Integration von heterogenen Applikationen. 1997, Universität St. Gallen: St. Gallen.
[18] Conrad, S., et al., Enterprise Application Integration - Grundlagen, Konzepte, Entwurfsmuster, Praxisbeispiele. 2006, Elsevier: Munich.
[19] Vogler, P., Prozess- und Systemintegration - Evolutionäre Weiterentwicklung bestehender Informationssysteme mit Hilfe von Enterprise Application Integration. 2004, Wiesbaden: Deutscher Universitäts-Verlag.
[20] Hasselbring, W., Information System Integration. Communications of the ACM, 2000. 43(6): p. 7.
[21] Gassner, C., Konzeptionelle Integration heterogener Transaktionssysteme. 1996, Universität St. Gallen: St. Gallen.
[22] Potts, C., Software-Engineering Research Revisited. IEEE Software, 1993. 10(5): p. 9.