Software Industrialization in Systems Integration

M.Minich[1,2], B.Harriehausen-Mühlbauer[2], C.Wentzel[2] and A.D.Phippen[1]

[1]Centre for Information Security and Network Research, University of Plymouth, Plymouth, United Kingdom
[2]Institute for Applied Informatics Darmstadt (aiDa), University of Applied Sciences Darmstadt, Germany
e-mail: matthias.minich@plymouth.ac.uk

Abstract

Today's economy is in a permanent change, causing merger and acquisitions and co operations between enterprises (Vogler, 2004). Consequential process adaptations and realignments usually result in systems integration and software development projects. Processes and procedures to execute such projects are still reliant on craftsmanship of highly skilled workers (Greenfield, 2004b). 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 particularities of systems integration. From the author's point of view it distinguishes itself from traditional software development in various points. The present work, and the subsequent research, will therefore focus on the implementation of industrialization concepts in the area of systems integration. The present paper briefly describes the idea of software industrialization, depicts current concepts from science, discusses the particularities of systems integration and suggests further areas of research. The objective of the suggested research should bring the area 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, Software Factories, Model Driven Engineering, Component Based Development.

1. Industrialization

Industrialization can be defined as the spreading of standardized and highly productive methods in production of goods and services in all economic areas (Brockhaus-Enzyklopädie, 1989; Butschek, 2007). The principle of 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 (Brockhaus-Enzyklopädie, 1989).

From a production point of view, omitting prerequisites such as the availability of resources and commodities or communication and transportation technologies, the key concepts of industrialization can be outlined as follows:

- Standardization
- Specialization
- Systematic Reuse
- Automation

These key concepts are often implemented in an "[…] organization of work known as the factory system, which entailed increased division of labour and specialization of function" (Encyclopedia Britannica, 1991). 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 automation as e.g. in the electronics industry, whereas creative tasks (which cannot be standardized), such as product design, are still performed by highly skilled workers.

2. Current concepts of industrialized software development

Software development is "[…] slow and expensive, and yields products containing serious defects that cause problems of usability, reliability, performance and security" (Greenfield and Short, 2004). It can be assumed that most of a program's functionality has already been developed in previous projects. If a consistent level of reuse and automation can be achieved, significant improvements in efficiency and quality can be made, which come along with noteworthy cost savings. In the following software development concepts are depicted which show signs of industrialization, as discussed in the previous chapter.

2.1. Model Driven Engineering

Model Driven Engineering aims to raise the level of abstraction of software engineering to fill the gap between the problem solution to be implemented and the actual technology utilized to do so. Once a suitable level of abstraction is found, the description of the solution has to be refined by adding previously omitted details until an executable implementation is available. The distance between the description and technical implementation characterizes what is commonly referred to as the abstraction gap.

Raising the level of abstraction has been researched on in the 1980s already with the upcoming of CASE-Tools. They encouraged development methods based on graphical representations of software with e.g. state machines, structure diagrams or dataflow diagrams (Schmidt, 2006) 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.

To overcome previously described difficulties, Model Driven Engineering (MDE) combines two important approaches: Domain Specific Languages (DSLs, also referred to as "Domain Specific Modelling Languages") and Transformation Engines and Generators (Schmidt, 2006), as described in the following.

2.1.1. Domain Specific Language

A Domain Specific Language (DSL) models concepts found in a specific domain, such as financial online services, e-commerce applications, CRM systems, or anything else clearly delimited. The characteristics of a specific domain are represented by metamodels, precisely specifying semantics and constraints associated with this particular domain (Schmidt, 2006).

> *"A modelling language is a visual type system for specifying model-based programs. It raises the level of abstraction, bringing the implementation closer to the vocabulary understood by subject matter experts, domain experts, engineers and end-users."(Greenfield and Short, 2004)*

One of the most successful examples of a Domain Specific Language can be found in WYSIWYG-Editors for graphical user interfaces. While in the beginning GUIs could only be built by highly skilled developers, today's wizards and code generators allow almost everyone to develop powerful user interfaces. What made this possible was the definition of a highly specialized, domain specific language, implemented in GUI design tools. Their elements (buttons, panes, text fields, etc.) can be combined based on clearly specified rules (e.g. Buttons can only appear within panes or windows etc.). Other well known examples are Event Driven Process Chains or the Entity Relationship Model (Beltran *et al.* 2007). With DSLs it should for instance be possible to assemble an online shopping system with credit approval, product catalogue and payment system without having to worry about the particular implementation and interaction of the components. To sum it up, DSLs have several important advantages (Beltran *et al.* 2007):

- Specifications can be described faster and more precise with DSLs
- Change requests can be captured precisely and unambiguously with DSLs
- Specifications are context free and leave no room for interpretations
- Code generators can be built for a specific domain and are thus more powerful and easier to handle as e.g. former CASE tools
- Transforming a model to source code by a generator is less error-prone than manual implementation for each product

2.1.2. Transformation Engines and Generators

Once a software system in a defined problem space has been specified with the help of the appropriate DSL(s), the thereby created set of models can be transformed to either intermediate models, or directly into source code. The former can be useful if the abstraction gap between a problem domain and the technical implementation capabilities is too large, e.g. if a specified system is supposed to run on different platforms - intermediate models would then take care of the particular requirements of these platforms. To generate subsequent artefacts out of models, transformation engines or code generators need to be provided together with meta-models of the source and target model, as well as a set of mapping rules between them. Whereas the meta-models are already available by the definition of the Domain Specific

Languages, the transformation rules must be expressed within a transformation language (Pham *et al.* 2007).

In their book "Software Factories", Greenfield and Short address Model Driven Engineering as one of the key innovations for software industrialization. They follow the differentiation of transformations as depicted by Czarnecki in (Czarnecki, 1999), which can be vertical, horizontal or oblique. Vertical transformations refine an existing model to a lower level, more concrete model or directly to source code. An example of vertical transformation is the transformation of a model describing a business process to a more detailed one, as for example the distribution over different web-services (Greenfield and Short, 2003). Horizontal transformations in contrast may either be refactoring or delocalizing transformations. "Refactoring transformations reorganize a specification to improve it's design without changing it's meaning", whereas "delocalized transformations can be used to optimize an implementation or to compose parts of an implementation that are specified independently" (Greenfield and Short, 2003). The former may for example adapt a model to a given architecture of a product line, whereas the latter may weave a security framework into the existing model.

2.2. Component Based Development

The idea of separating software into delimited parts out of which applications can be stitched together as needed, is probably as old as software development itself. It first appeared in literature at the NATO Software Engineering Conference where M.D. McIlroy suggested that we need a software component sub industry, "available in families arranged according to precision, robustness, generality and time span performance" (Software Engineering, 1968). In his book about component software (Szyperski, 1998), Szyperski defines 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".*

A component requires a defined environment and interacts with this environment via defined interfaces, without revealing the actual implementation of the functionality it provides. Ideally, components are language neutral and neither platform constraint, nor application bound. Based on Brown (Brown and Wallnau, 1996) and adapted by Haines and Foreman (Haines *et al.* 1997), component based development can be subdivided into four major steps:

During the first step (component qualification) existing components are discovered and evaluated against their potential to be deployed in another context. The result of the qualification defines whether certain functionality can be integrated from existing artefacts or must be manually developed. Component qualification may include functional and non-functional requirements such as algorithms or interfaces and quality or performance.

If required, suitable components can be adapted in the next step. Adaptations could be wrappers for underlying platforms or the integration of certain aspects as e.g. security concepts. Components can be categorized into white-box, grey-box and black-box ones (Haines *et al.* 1997). The former allow significant changes to the component at the cost of compatibility and replace ability. Adaptations to the latter have very little negative side effects, but may not allow the required flexibility. Grey-Box components do not allow changes to their source code but provide extension languages or APIs (Haines *et al.* 1997) to adapt them to specific requirements.

In a third step the previously qualified and adapted components are assembled to a new application. This assembly is usually built on frameworks which provide the implementation base for the components. „It is therefore very important that there exist a context in which […] *[components]* can be used“ (Crnkovic *et al.* 2002). Frameworks furthermore overlap with patterns, which „[…] define a recurring solution to a recurring problem“ (Crnkovic *et al.* 2002).

The final step focuses on maintenance and enhancement. Components are replaced with their improved or debugged versions or with totally new ones, combining the functionality of multiple already existing ones (Haines *et al.* 1997).

2.2.1. Current implementations and frameworks

The IT landscape provides several implementations of component based development, which are primarily concerned with the technical mechanisms of enabling components to communicate with each other. The most prominent representatives are CORBA, COM/DCOM, Web Services and EJBs:

- **CORBA:** The Common Object Request Broker Architecture (CORBA) defines a standardized model for inter-component communication and defines specific operations which describe the collaboration of distributed systems. The central concept of CORBA is the Object Request Broker (ORB), through which different components communicate with each other (Lexikon der Kommunikations- und Informationstechnik, 2001). It takes requests, locates the required component and forwards the request transparently. Interoperability between languages is ensured by an Interface Definitions Language to describe the external boundaries of a component in a standardized way (Computer und Informationstechnologie, 2005), and language specific ORB implementations.

- **COM/DCOM:** The Distributed Component Object Model is an architecture developed by Microsoft for the communication between components, based on a Windows operating platform. It uses proxies, providing interfaces and stub code by abstract methods and memory pointers (Computer und Informationstechnologie, 2005). They can be seen as a virtual substitute, forwarding requests to the actual component. Communication within a single computer system occurs directly through shared memory, for distributed systems it occurs via Remote Procedure Calls (distributed COM

or DCOM). Interfaces of components are described with the Microsoft Interface Definition Language (MIDL). To allow interoperability between different programming languages, any data is converted into a normalized format. However, support for proxies and component discovery is primarily available for windows platforms. (Computer und Informationstechnologie, 2005).

- **Web-Services / SOA:** One of the most recent approaches is depicted by the Service Oriented Architecture (SOA). It aims at the provision of business functionality as clearly delimited services in order to avoid the "[…] duplication of code […] for enabling similar business functions across multiple business processes, spanning one or more lines of businesses" (Dan *et al.* 2008). Ideally, services are delimited, available in a network, have published interfaces, are platform independent, and registered in a repository. The most prominent implementation, Web Services, is based on three major concepts: Universal Description Discovery and Integration (UDDI) for component registration and indexing, Web Service Description Language (WSDL) for a precise, XML-based description of the supported functionality, methods and parameters, and the Simple Object Access Protocol (SOAP) for XML-based communication between service consumer and provider, encapsulated in common internet protocols such as HTTP for example.

- **EJBs:** Enterprise Java Beans is a component oriented framework for distributed information systems. It is based on the J2EE library and allows the invocation of remote methods via Web Services, IIOP (Internet Inter ORB Protocol, based on CORBA), Native Java, or JMS (Java Message Service). While Java and JMS require a Java implementation on both sides of the connection, Web Services and IIOP allow platform independent communication between different components. However, the EJB concept does not offer any transformation of data.

While MDE or CBD were successfully introduced in smaller areas, different technologies, platform dependencies or high initial investments anticipated the successful integration of already existing components into new applications across-the-board. Apart from Web Services, CORBA can be seen as the dominant model for a corporate wide system landscape as it is platform and language independent and an open standard, already in its third generation (Lewandowski, 1998; Schryen, 2001). Despite some experimental adaptations, COM/DCOM relies on concepts of the Microsoft Windows platform, which prevent it from being adopted by major business software providers which usually offer their products on different platforms (e.g. UNIX). The EJB concept may be platform independent, but relies on Java implementations on both sides. From a market perspective, only CORBA and COM/DCOM have enough momentum, supplier support and a large enough feature set to serve as long term technologies (Lewandowski, 1998).

Web Services in contrary focus more on the distributed and software-as-service aspect and offer their services platform and language independent over networks.

The author therefore expects CORBA to become the major concept for component oriented and distributed, but company wide system landscapes. Web services may find their focus in specific services offered by external providers over the internet, such as credit approval by financial institutions for example.

2.3. Software Product Lines and Software Factories

Greenfield and Short suggest the term "economies of scope" to describe the basic principle of software industrialization. "Economies of scope arise when multiple similar but distinct designs and prototypes are produced collectively, rather than individually" (Greenfield and Short, 2004). Economies of scale in contrast arise when several copies of exactly the same product are created. As this can be done very easily with software, economies of scale do not offer any advantages. Economies of scope can be compared to a car manufacturer for example. Besides model specific body parts, car makers mostly assemble their cars from standardized components like engine blocks, gearboxes or electronic control units. Depending on the customer's wishes, specific components are selected and assembled to a complete car. Most of the components may also be used in another model.

The concept of software product lines requires to separate product development from product line development. The former produces the actual software product, while the latter produces all the required assets to support the product development process. The concept furthermore groups closely related products to a product family. This has the advantage that the assets of a product line are more specific and powerful to a problem than generic concepts could be. "A software product line systematically captures knowledge of how to produce the assets, such as components, processes and tools, and then applies those assets to produce the family members" (Greenfield and Short, 2004).

In their book "Software Factories", Greenfield and Short take the software product line approach one step further by introducing the concept of software factories.

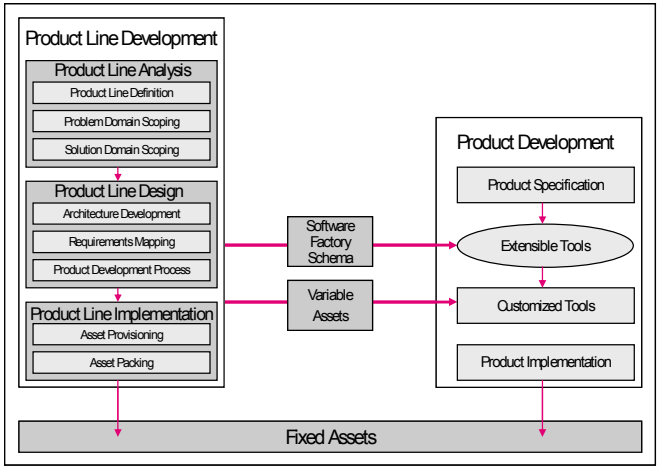

Figure 1: A Software Factory

It proposes a way to "categorize and summarize development artefacts, such as XML documents, models, configuration files, build scripts, source code files, […], in an orderly way […]" to define relationships and dependencies among them (Greenfield and Short, 2004). Given a certain product, the software factory identifies the required artefacts and assets of the respective product line in order to develop the product. It does so by defining software factory schemas which exactly define which assets like micro-processes, frameworks, architectures and tools or domain specific languages, are to be used to produce a family member, as illustrated in the previous figure.

To implement the idea of software product lines and software factories, Greenfield and Short demand the further development of four critical innovations (Greenfield and Short, 2004):

- **Systematic Reuse**: Technologies like CORBA, J2EE or COM/DCOM offer the basic principles required for reuse. However, the main problem with such technologies is the lack of a specific context. Components are too generic to cover all possible implementation scenarios in arbitrary contexts (Greenfield and Short, 2004). Components developed in a specific context may be reused more easily in a similar context. A component used for payment verification can be much more powerful if it is only used within the context of web based applications and not within mainframes as well.

- **Development by Assembly**: This critical innovation subsumes five prerequisites required to support development by assembly. *Platform independent protocols* to avoid interoperability problems between components. *Self description* (or contracts), including assumptions, dependencies and behaviour, allow for proper selection and validation of assemblies. To be able to customize, a *deferred encapsulation* of existing components is necessary, which "[…] reduces architectural mismatch by waving adaptations into published components" (Greenfield and Short, 2004). To reduce the risk of architectural mismatch, *architecture driven development* must be implemented by imposing assumptions and constraining design decisions. Similar to web-services, the fifth prerequisite suggests the *assembly of components by orchestration*. The latter can be seen as an automated combination and functional management of independent components.

- **Model Driven Development**: Further automation of software development requires formal specifications in a way humans and machines can understand. Thus MDD uses formalized models to precisely capture developer intend. The models can then be used to either refine or transform the requirements to a more detailed layer or to generate code artefacts out of them. Formalized models can be expressed in a Domain Specific Language, which is designed for an explicit purpose such as a software product family. "A well-defined DSL is a powerful implementation language, providing much greater rigor than a general purpose modelling language like UML" (Greenfield and Short, 2004). Additional improvements can be achieved if

the abstractions of the model are used to generate a framework which guides the developer in completing the application.

- **Process Frameworks**: As with components, many process frameworks are too abstract and thus require rethinking about how to apply a process to a specific task. More specialized processes centre on the development of assets of a product within a software product line. Further gains in productivity can be achieved by integrating these processes into development environments, guiding and constraining the developers in their work. (Greenfield and Short, 2004)

As can be seen from the previously mentioned four critical innovations, the concept of software product lines, and software factories respectively, combines prevailing concepts like Model Driven Development (which can be seen as a part of Model Driven Engineering), Component Based Development and Software Reuse technologies into a holistic approach of software industrialization.

3. Particularities of Systems Integration

The field of systems integration (SI) comes with several particularities, distinguishing it from the domain of conventional software development. Systems integration has to challenge a multiplicity of technologies, once only technology combinations and a very high complexity of to be integrated systems. According to Vogler in (Vogler, 2004), potential problems can furthermore be categorized as follows:

Problem Area	Problems
Know-How	Lack of knowledge about potential solutions Unknown consequences of integration decisions
Management	Suboptimal degree of integration Unknown integration relationships High time pressure within the integration project No methodical approach Unknown complexity of the project Lack of standards
Information systems	Heterogeneity of systems to be integrated Lack of flexibility in legacy systems Data redundancy within different systems

Table 1: Integration problems and problem areas

3.1. Know-How related

The problem area related to know how issues, embraces the lack of knowledge about potential solutions for a given problem. The multiplicity of different systems and technologies make it difficult for system engineers to select the optimal implementation. It is for example very unlikely that an expert for Siebel CRM Systems will also be an expert for SAP. Besides the technical implementation, it is

also necessary to consider the pivotal business process during integration (Vogler, 2004). Furthermore, companies may not be aware of solutions and products available on the market and may not be able to develop integrated concepts for their IT landscape.

Another know-how related problem is the uncertainty of consequences if a system in a highly integrated environment is altered. This becomes especially evident as systems integration often occurs on a per project basis, implementing merely the prevailing requirements without aiming at a company wide integration concept. This may lead to n*(n-1) relationships between different systems and thus requires a very careful consideration of affected systems before conducting a change.

3.2. Management related

One problem is the suboptimal degree of integration. According to (Vogler, 2004), two extremes can be found: isolated applications or highly integrated ones with peer-to-peer characteristics. The former is usually specialized in a particular task, not providing any interfaces to link it to other systems. While the former is hard to integrate, the latter is tightly interwoven with the IT landscape. Only very few enterprises consistently use a common architecture like a messaging middleware for example (Longo, 2001; Vogler, 2004).

Unknown integration relationships impose a problem on ad-hoc changes to information systems. Short and simple workarounds to quickly fix a problem may not be documented and thus remain unconsidered for potential changes. This lack of transparency prevents completeness and consistency checks for interfaces (Vogler, 2004).

High time pressure within the integration project may lead to the omittance of documentation and testing. Unfortunately both are crucial in an integrated environment as other systems rely on the interface descriptions and a credible service provisioning. However, a trade-off must be found between the efforts put into documentation and the benefits it generates.

To solve complex problems in software engineering, methodologies are being used (Heinrich *et al.* 2004) such as the Rational Unified Process or V-Model XT. While it is performed for years now, still no generally accepted methodology or approach for systems integration has been found (Vogler, 2004; Engel, 2006). This shortcoming is assumed to origin from the fact that integration is often seen as a purely technical problem which has to be resolved after completion of the underlying systems (Gassner, 1996).

Unknown consequences of changes to the IT landscape, unknown integration relationships or highly interweaved systems lead to a very high complexity which may remain unidentified, thus leading to increased cost and time to complete.

Similar to the previously depicted missing methodologies, systems integration also lacks generally accepted standards. This shortcoming is caused by the heterogeneity

of applications (Vogler, 2004) and the fact that a prospective integration is unforeseeable during the development of applications. However, recent work in the field of Enterprise Application Integration (EAI) has developed fist concepts and frameworks, usually based on interapplication middlewares or Service Oriented Architectures, as for example in (Lee *et al.* 2003; Gorton and Liu, 2004; Sutherland and van den Heuvel, 2002; Strüver, 2006).

3.3. Information Systems related

One of the core issues or particularities of systems integration is the heterogeneity of to be integrated systems (Longo, 2001; Stickel, 2001; Riem, 1997). Differences can not only be found on a technical layer (programming languages, operating systems) but also on a logical and conceptual layer (system architecture, frameworks, data structures) (Vogler, 2004). Both layers have a major influence on an adequate integration and thus need to be considered when implementing industrialization concepts in the field of systems integration. This heterogeneity anticipates the formation of standards as e.g. company wide integration architectures, which in turn leads to a discontinuity of media (media disruption). Furthermore, heterogeneity is reinforced by the fact that integrated systems are usually connected on a peer-to-peer basis with each other, leading to n*(n-1) relationships. As of the high costs of enterprise information systems, applications are usually not replaced frequently. "[…] SI aims at building applications that are adaptable to business and technology changes while retaining legacy applications and legacy technology as long as possible" (Hasselbring, 2000). This disadvantage further complicates systems integration due to insufficient reusability, outdated data management and user interfaces, monolithic constructions or inadequate maintainability (Vogler, 2004).

If different applications are merged into an integrated system, data and even functional redundancy may occur. Unless one data storage is a definite master or synchronization takes place, each transaction has to ensure that it works with the most actual data to prevent data inconsistency. In addition to the syntactical consistency of data, their semantics must also be ensured across different applications.

4. Industrialization in Systems Integration

The focus of the intended research is aimed at the application of industrial production principles in the specific domain of systems integration. As described in chapter 3, systems integration differs in certain areas from the development of traditional software products. Especially the heterogeneity of products is one of the major differences (Longo, 2001; Stickel, 2001; Riem, 1997), which in turn leads to the question whether currently discussed industrialization concepts are suitable.

4.1. Software Factories in regard of SI particularities

The present section will briefly discuss the SI particularities (q.v. Table and chapter 3) in context of industrialized software development. It thereby centres around the idea of Software Factories, as the underlying concept comprises Component Based

and Model Driven Development (as part of Model Driven Engineering), together with Software Product Lines, code generation and systematic reuse (Greenfield and Short, 2004). From the author's point of view, Software Factories is currently the most advanced and comprehensive concept of software industrialization. Software Factories concentrate on the following four critical innovations to be introduced.

4.1.1. Systematic reuse

Greenfield & Short (Greenfield and Short, 2004) suggest to partition software engineering efforts into clearly delimited product lines. In doing so, design and development occur in a particular context, sharing common features and solving common problems conjointly. Product families may either be tailored around complete products or a series of related components. They concentrate on reusable implementation artefacts, as well as frameworks, processes and tools.

> *"Program families enable a more systematic approach to reuse, by letting us identify and differentiate between features that remain more or less constant over multiple products and those that vary" (Greenfield and Short, 2004).*

With reference to the particularities of 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 all eventualities by supporting 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. Barely any development 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.

4.1.2. Development by Assembly

The second critical innovation is the logical consequence of systematic reuse. According to Greenfield & Short (Greenfield and Short, 2004), development by assembly itself has certain requirements which 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 new aspects), assembly by orchestration (machine controlled interaction and management of components), and architecture driven development (to promote the availability of well-matched components) (Greenfield and Short, 2004). The latter is seen to be most critical for development by assembly.

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,

which may not be easy to define. 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 as e.g. service oriented architectures are not common until now (Lee *et al.* 2003; Gorton and Liu, 2004; Sutherland and van den Heuvel, 2002; Strüver, 2006). The most important challenge to be met is the definition of software architectures and standards in which development by assembly may occur.

4.1.3. Model Driven Development

Model Driven Development, and in a greater sense Model Driven Engineering, raises the level of abstraction to alleviate increasing complexity and expressing domain concepts more efficiently (Schmidt, 2006) and context free. It consists of domain specific modelling languages, along with transformation engines and generators. The former allow a powerful description of the intended products of a product line, whereas the latter provide model transformation to a lower, more specific layer or eventually the generation of source code.

With regard to systems integration, the efforts required to define a domain specific language could become an obstacle, especially if applied to very small product lines. Furthermore, to automate the development process by generating source code or transforming models to a lower level, transformation engines and code generators have to be implemented. As directly related to domain specific languages, they are also product line specific. The integration aspect itself may be an additional challenge. Domain specific languages, models and architectures have to be compatible between the product lines whose products are to be integrated with each other.

4.1.4. Process Frameworks

“The key to process maturity is preserving agility while scaling up to high complexity created by project size, geographical distribution, or the passage of time” (Greenfield, 2004a). While process frameworks like RUP, XP or Waterfall XT are widely available, Greenfield & Short (Greenfield and Short, 2004) demand an extensive customization of development processes to balance cumbersome formalism and agility. Depending on the selected product line features, the process framework can be further customized to support the development of the actual software. In a subsequent step, the process definitions may be incorporated into development tools, providing active guidance to the developer without hindering agility.

Yet again it comes back to clearly delineating a specific context, which is currently not given in the domain of systems integration. As with model driven development, process frameworks also need to be compatible to each other between different product lines in order to simplify integration. The incorporation of process definitions and process imposed restrictions or boundaries into development tools is a requirement which can hardly be solved by software development companies. The author assumes this to be subsequently solved by tool suppliers as software

industrialization advances and becomes more accepted as a new development paradigm. It is therefore beyond the scope of the intended research.

4.2. Areas requiring further research

As can be seen in the previous section, 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. Out of the previous sections, certain key questions arise, which will be briefly discussed in the following.

4.2.1. Organizational aspects

Organizational aspects focus on the surrounding conditions of industrialization in SI. They should be carefully considered before performing a paradigm shift throughout the organization.

1. *How can we define areas of specialization in systems integration, considering the multiplicity of different technologies and their rare combinations within integration products?*

The definition of narrow and clearly delimited problem domains seems inevitable for an industrialized production. With regard to systems integration, how can we carve out the combination of business domain knowledge with a multiplicity of different technologies? What is a reasonable organizational structure and how can we ensure that integration requirements can still be mapped onto the new organizational structure?

2. *How can we measure the degree and success of software industrialization in systems integration?*

In large organizations, efficient steering mechanisms are required. Conventional software engineering provides measures like *function points per time unit* for productivity or *defects per function point* for quality. But what are reliable measures to manage and monitor an industrialized production? Can we develop something like an Industrialization Maturity Model, similar to CMMI for example?

4.2.2. Technological aspects

The technological aspects focus more on the actual implementation of critical innovations and key concepts within the context of systems integration.

3. *Can we apply essential innovations of software industrialization to delimited problem domains within systems integration?*

Given that an expedient classification of activities into e.g. product lines or services has taken place, can we still apply the essential innovations such as systematic reuse,

development by assembly, model driven development and specialized process frameworks?

4. *Are these essential innovations suitable for all application domains within systems integration, such as SAP, Siebel or PeopleSoft?*

Many projects in the field of systems integration include development work for more sophisticated IT systems such as SAP for example. As these systems are often customized by using graphical development tools, how do concepts like component oriented or model driven development / engineering fit?

5. *Which preconditions must be met to automate e.g. model transformation and code generation?*

The probably most ambitious objective of an industrialized software development is the automated creation of artefacts such as model transformations to more detailed models or source code generation. Playing into organizational aspects as well, how high is the effort to implement such a concept? Do we need separate tools for each problem domain or can we reuse their foundation?

4.2.3. Integrative aspects

The following research questions are closely related to technological aspects, as they discuss the interoperability of product domains between organizations.

6. *How can we ensure compatibility between domain specific tools and assets of different problem domains?*

Assumed key question 3 has successfully been answered and the critical innovations are implemented in clearly delimited problem domains, how can we ensure that we still can combine a multiplicity of technologies in an integration product? Are Domain Specific Languages compatible to each others or can we fit components of problem domain A into the framework of problem domain B? Systems usually need to be planned and designed in a holistic approach (at least on a coarse level).

7. *Can industrialized systems integration be aligned along broadly accepted standards in the field of systems integration?*

As discussed in chapter 3, systems integration lacks standards and methodical approaches and thus suffers from high heterogeneity. Is it reasonable to align the industrialization concept on broadly accepted standards (if available) in order to alleviate such problems in the future?

5. Summarization and Outlook

Systematic reuse of existing software artefacts 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 Model Driven Development, Component Based Development, and Software Product Lines, are currently being developed and implemented, as described in chapter 2. The proposal of Software Factories by Greenfield & Short (Greenfield and Short, 2004) combines new and already existing concepts into a holistic approach of software industrialization. However, as software engineering takes place in a wide variety of application domains, we cannot be sure 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 business processes or business requirements. To better understand the particularities of this field, chapter 3 depicts its substantial differences. Consequently, chapter 4 discusses the suitability of existing concepts with reference to systems integration and identifies the following difficulties and shortcomings:

- A high heterogeneity in the projects of a systems integrator prevents the traditional implementation of software product lines, unless they are exceptionally narrow.
- Diverse technologies in a product family prevent building up technical expertise. Dedicated (technical) development teams per product line don't seem to be viable.
- The implementation efforts for setting up and maintaining the previously described "critical innovations" in small software product lines may consume potential savings.
- Organizational aspects and requirements of software industrialization in systems integration are yet unknown, especially with respect to the previously described difficulties and shortcomings.
- The lack of standardized frameworks and architectures in the field of systems integration may prevent an industrialized collaboration between enterprises, e.g. to form a software supply chain.

Section 4.2 subsequently identifies further areas of research and categorizes them into organizational, technological and integrative aspects. The first category is concerned with the future organizational structure of a systems integration organization, in respect of clearly delimited problem domains. Technological aspects cover the actual implementation of technical concepts, their suitability for particular areas, and preconditions for an increased level of automation. The final category, integrated aspects, deals with the compatibility of industrialized development methods across problem domains.

The present paper outlines particularities and potential challenges of industrialized systems integration, as well as further areas of research to get there. In order to pursue a structured approach and as some research topics depend on the answers of others, the author suggests the following redefined order and consequential structure of the research project, based on the key questions (KQ) in section 4.2:

- ***KQ 1***: Elaboration of an organizational structure for industrialized systems integration with reference to the specialization in a heterogeneous environment, as well as (anticipating KQ 4) the application of critical innovations as depicted in section 4.
- ***KQ 3***: Evaluation of the applicability of essential innovations as e.g. component oriented or model driven development to delimited problem domains of systems integration. This question should also bear cost and return on investment in mind.
- ***KQ 6***: Analyse the interoperability of assets derived from different problem domains in order to support the fundamental concept of systems integration.
- ***KQ 2***: Once the most fundamental concepts and questions are in place and answered, develop measures and metrics representing the degree and success of industrialization.
- ***KQ 5***: With regard to the size of potential problem domains, identify the preconditions and efforts incurred with automated model transformation and code generation (if applicable).
- ***KQ 4***: Identify problem domains with more sophisticated products and development tools such as SAP or Siebel and evaluate the applicability of software industrialization concepts in these particular areas.
- ***KQ 7***: Provide an outlook on the interoperability of industrialized system integrators with regard to generic standards and frameworks in the field of systems integration.

The above depicted further research on major problems of industrialized systems integration will be conducted in close collaboration with representatives of the industry to obtain first hand experiences and validate the results of the latest research in practice. The obtained results of the particular problems will be presented within scientific papers and conference contributions, whereas the concluding dissertation will draw a holistic picture of industrialized systems integration and demonstrates methods and techniques to get there.

6. References

Beltran, J.C.F., Holzer, B., Kamann, T., Kloss, M., Mork, S.A., Niehues, B. and Thoms, K. (2007), *Modellgetriebene Softwareentwicklung*, Frankfurt, entwickler.press.

Brockhaus-Enzyklopädie. (1989), Brockhaus-Enzyklopädie. 19 ed. Mannheim, F.A. Brockhaus.

Brown, A.W. and Wallnau, K.C. (1996), *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*, Los Alamitos, IEEE Computer Society Press.

Butschek, F. (2007), *Industrialisierung*, Ulm, Ebner & Spiegel.

Computer und Informationstechnologie. (2005), In Greulich, W. (Ed. Der Brockhaus. Leipzig, Mannheim, F.A. Brockhaus GmbH.

Crnkovic, I., Hnich, B., Jonsson, T. and Kiziltan, Z. (2002), "Specification, Implementation, and Deployment of Components", *Communications of the ACM*, 45, 6.

Czarnecki, K. (1999), "*Generative Programming - principles and techniques of software engineering based on automated configuration and fragment-based component models*", Illmenau, Technische Universität Illmenau.

Dan, A., Johnson, R.D. and Carrato, T. (2008), "SOA Service Reuse by Design", *International Conference on Software Engineering* Leipzig, ACM.

Encyclopedia Britannica (1991), Encyclopedia Britannica. 15 ed. Chicago, Encyclopedia britannica inc.

Engel, T. (2006), "*Ein Beitrag zur unternehmensübergreifenden Integration von Informationssystemen*" Institut für Rechneranwendung in Planung und Konstruktion. Karlsruhe, Universität Karlsruhe.

Gassner, C. (1996), "*Konzeptionelle Integration heterogener Transaktionssysteme*", St. Gallen, Universität St. Gallen.

Gorton, I. and Liu, A. (2004) "Architectures and Technologies for Enterprise Application Integration", *International Conference on Software Engineering*. Edinburgh, IEEE.

Greenfield, J. (2004a) "Problems and Innovations - Building Distributed Applications", *Microsoft Architect Journal*.

Greenfield, J. (2004b) "Scaling Up Software Development", *Microsoft Architect Journal*.

Greenfield, J. and Short, K. (2003) "Software Factories - Assembling Applications with Patterns, Models, Frameworks and Tools", *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Anaheim, USA, ACM.

Greenfield, J. and Short, K. (2004) *Software Factories - Assembling Applications with Patterns, Models, Frameworks, and Tools*, Indianapolis, John Wiley & Sons.

Haines, G., Carney, D. and Foreman, J. (1997) "*Component Based Software Development / COTS Integration*", Pittsburg, Carnegie Mellon University.

Hasselbring, W. (2000) "Information System Integration", *Communications of the ACM*, 43, 7.

Heinrich, L., Heinzl, A. and Roithmayr, F. (2004) *Wirtschaftsinformatiklexikon*, 7 ed. München, Wien, Oldenbourg.

Lee, J., Siau, K. and Hong, S. (2003) "Enterprise Integration with ERP and EAI", *Communications of the ACM*, 46, 7.

Lewandowski, S. M. (1998) "Framworks for Component-Based Client/Server Computing", *ACM Computing Surveys*, 30, 25.

Lexikon der Kommunikations- und Informationstechnik. (2001) In KLUßMANN, N. (Ed. Lexikon der Kommunikations- und Informationstechnik. 3 ed. Heidelberg, Hüthig Verlag.

Longo, J. (2001) "The ABCs of Enterprise Application Integration", *EAI Journal*, 3.

Pham, H.N., Mahmoud, Q.H., Ferworn, A. and Sadeghian, A. (2007) "Applying Model-Driven Development to Pervasive System Engineering", *International Conference on Software Engineering*. Minneapolis, IEEE Computer Society.

Riem, R. (1997) "*Integration von heterogenen Applikationen*", St. Gallen, Universität St. Gallen.

Schmidt, D. C. (2006) "Model Driven Engineering", *IEEE Computer*, 39, 7.

Schryen, G. (2001) *Komponentenorientierte Softwareentwicklung in Softwareunternehmen: Konzeption eines Vorgehensmodells zur Einführung und Etablierung*, Wiesbaden, Deutscher Universitäts-Verlag.

Software Engineering. (1968) in Naur, P. and Randell, B. (Eds.) *NATO Software Engineering Conference*. Garmisch Partenkirchen, NATO Science Committee.

Stickel, E. (2001) "*Informationsmanagement*", München, Wien, Oldenbourg.

Strüver, S.-C. (2006) "*Standardisiertes EAI-Vorgehen am Beispiel des Investment Bankings*", Berlin, GITO.

Sutherland, J. and Van Den Heuvel, J. (2002) "Enterprise Application Integration and Complex Adaptive Systems", *Communications of the ACM*, 45, 6.

Szyperski, C. (1998) *Component Software: Beyond Object-Oriented Programming*, Massachussetts, Addison-Wesley.

Vogler, P. (2004) "*Prozess- und Systemintegration - Evolutionäre Weiterentwicklung bestehender Informationssysteme mit Hilfe von Enterprise Application Integration*", Wiesbaden, Deutscher Universitäts-Verlag.