Model Driven Engineering in Systems Integration

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

[1]Centre for Security, Communications and Network Research,
Plymouth University, Plymouth, UK
[2]Department of Computer Science, University of Applied Sciences Darmstadt,
Darmstadt, Germany
e-mail: info@cscan.org

Abstract

Software development in systems integration projects is still reliant on craftsmanship of highly skilled workers. To make such projects more profitable, an industrialized production, characterized by high efficiency, quality, and automation seems inevitable. While first milestones of software industrialization have recently been achieved, it is questionable if these can be applied to the field of systems integration as well. Besides specialization, standardization and systematic reuse, automation represents the final and most sophisticated key concept of industrialization, represented by Model Driven Engineering (MDE). The present work discusses the most prominent MDE approaches, while considering the particularities of systems integration. It identifies Generative Programming as being most suitable and integrates it into previous works on Software Product Lines and Component Based Development in Systems Integration.

Keywords

Software Industrialization, Automation, Systems Integration, Software Product Lines, Generative Programming, Model Driven Engineering

1. Introduction

Compared to other high tech industries, software engineering shows only marginal improvement in terms of productivity, quality, and cost efficiency. It is still characterised by a high degree of craftsmanship to develop software from scratch with labour-intensive methods. By applying industrial methods and thus enhancing an organization's productivity, we possibly can increase quality and product complexity, and at the same time reduce cost and production time. Key industrial methods can be defined as specialization, standardization, systematic reuse, and automation (Encyclopaedia 2005). In the field of software engineering, Software Product Lines (SPL) represent specialization as the first and probably most important industrial principle. By concentrating on a limited scope, production assets can be much more power- and useful, which is especially important for standardization and systematic reuse as the second industrial principle. Both are available within Component Based Development (CBD), an approach to exchange and systematically reuse software artefacts in a standardized fashion. The final aspect of industrialization, automation, can be achieved with Model Driven Engineering (MDE). Using models as a description of software and utilizing domain specific languages, the degree of freedom and possible contexts available to a software developer is reduced. Without such limitation it would hardly be possible to provide formal model transformation engines and code generators, as they would have to cover an indefinite number of possible implementations for e.g. a single business concept.

In today's business world, IT faces high demands in quickly adopting to new 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. This situation inevitably leads to systems integration efforts, joining the different subsystems into a cohesive whole, in order to provide new business functionality or data access (Fischer, 1999; Leser and Naumann, 2007). Systems integration deals with the steps required to move an IT system from a given degree of integration to a higher one by merging distinct entities into a cohesive whole, or integrating them into already existing systems (Riehm, 1997; Fischer, 1999).

Although several literature on the different industrialization concepts and their practical implementation is available (Clements and Northrop, 2007; Herzum and Sims, 2000; Stahl and Bettin, 2007), it seems questionable if they are suitable for all areas of software development, such as systems integration with its high heterogeneity or single-use development projects. The present work therefore takes the position of a large systems integrator, who provides enterprise application integration (EAI) services and solutions to his customers. Research was done with support of a German company active in the field, providing a variety of integration solutions to its customers. The objective was to identify different possibilities for model driven engineering while considering the particularities of the company: Taking into account that such providers are usually involved in different industries; a high heterogeneity must be assumed. This anticipates the formation of standards and is reinforced by the fact that integrated systems are often connected on a peer-to-peer basis with each other. Due to high acquisition cost, they are also not replaced frequently (Hasselbring, 2000).

It must be assumed, that for such heterogeneous, volatile, and customer specific projects, conventional industrialization approaches are hardly feasible. For Software Product Lines and Component Based Development, we have developed a methodology in our previous works about an Organizational Approach for Industrialized Systems Integration (Minich *et al.*, 2010), and Component Based Development in Systems Integration (Minich *et al.*, 2011). The present work deals with the implementation of the third and final industrial key principle, i.e. automating development with the help of Model Driven Engineering. With the given situation and existing MDE concepts, it must be assumed that such intent will never break even, as no considerable economies of scale or scope exist to justify expenses for domain specific language, transformer, and generator development. To overcome this challenge, either reusability or cost efficiency must significantly be increased.

2. Automating Software Development

In automated software development, software engineers specify what to do, but not how. It is up to model transformers or code generators to interpret descriptive models of the intended system and create either intermediate models to be further refined, or source code. Different approaches exist or are currently being researched. The following are the most discussed ones in literature:

- Model Driven Architecture (MDA): An initiative from the Object Management Group (OMG), MDA defines a model driven development approach which is based on a separation of functional and technical concerns (Object Management Group, 2003). It therefore specifies UML as its modelling language, and the Meta Object Facility as its describing model (meta model) for all specification models. These are the Computation Independent Model (CIM), the Platform Independent Model (PIM), the Platform Specific Model (PSM), and the Platform Specific Implementation (PSI). The CIM describes the required systems from hard- and software independent point of view. It can be represented as a high level UML class diagram containing the key concepts and terms of the respective domain. The CIM is further elaborated with conceptual information and transforms into a PIM, describing the required system on a formal and precise level, containing elements like entities, attributes, or data types (Petrasch and Meimberg, 2006). The PIM is the first model which may automatically be transformed by transformation engines or code generators and thus needs to be as precise as possible (Singh and Sood, 2009). Subsequently, it is transformed into the PSM, formally describing the application for the specified platform. Several iterations are possible, until the final result is the Platform Specific Implementation, i.e. an executable artefact reflecting the requirements previously depicted in the CIM.
- Generative Programming (GP): Based on the work of Czarnecki and Eisenecker (Czarnecki, 2005), Generative Programming aims at automating the development of a family member within a Software Product Line. It therefore defines a problem space expressed by a Domain Specific Language and the solution space consisting of "implementation-oriented abstractions, which can be instantiated to create implementations of the specifications expressed using the domain-specific abstractions from the problem space" (Czarnecki, 2005). The mapping between both contains the configuration knowledge such as illegal feature combinations, default settings, default dependencies, construction rules and grammar, or optimizations. These mapping rules are implemented within a generator returning the solution space, which may either be an intermediate model or executable program code.
- Software Factories (SF): An approach introduced at Microsoft by Greenfield and Short (Greenfield *et al.*, 2004) which, similar to GP, utilizes Software Product Lines and Component Based Development, along with a highly customized IDE. It is based on Software Factory Schemes, which describe certain viewpoints required to develop a system. Such viewpoints express concerns regarding the business logic and workflows, data model and data messaging, application architecture, and technology, and may be present on all levels of abstraction. All together the schemes with their viewpoints exactly define what needs to be done and how to manufacture a family member. In order to provide a customized IDE, the schema with its viewpoints is represented by a Software Factory Template. The template can be loaded into an IDE, providing wizards, patterns, frameworks, templates, domain specific languages, and editors. Complete definitions of domain specific languages furthermore allow (semi-) automatic model to model transformations and code generation.

Compared to MDA, Generative Programming has a domain oriented focus which is usually found in Software Product Lines. MDA in turn does not necessarily rely on a clearly delimited problem domain. GP furthermore allows to create DSL, generator, and other artefacts required “on the fly” during regular software development. This reduces the necessity of high upfront investments and leads to artefacts tailored exactly to the needs of the implementing company. In contrast to GP and MDA, Software Factories are currently based on proprietary IDEs and modelling frameworks from Microsoft. Furthermore, most of the infrastructure needs to be in place before software development may start, leading to high upfront investments. Comparing MDE with previous advances of software development, such as compilation technology or 3rd generation languages, further advancing the level of abstraction and thus increasing automation seems obvious.

However, even after almost 30 years of research in Computer Aided Software Engineering (CASE) and similar approaches as the ones introduced above, this has not yet happened. In an article on automation and model based software engineering (Selic, 2008), Bran Selic names some of the most significant reasons for the lack of acceptance of automated software development in the industry. Foremost, the biggest advantage of fourth generation programming languages (i.e. Domain Specific Languages) is also their biggest drawback: A limited scope makes them very powerful, but also reduces the economies of scale for any infrastructure development such as IDEs, transformation engines, or code generators. Development tools are either built in-house and commercially hardly break even, or by a very small number of vendors, leading to a vendor lock-in. In addition, software developers sufficiently skilled in a particular language or toolset are highly specialized and not easily available on the market. However, even with such available, there are still some more pragmatic issues such as usability of large graphical models, interoperability between tools, or current development culture (Selic, 2008).

In conclusion it can be said that with Model Driven Architecture, Generative Programming, and Software Factories, there are some interesting and promising approaches being developed. However, their way into industrial practice is still prone to “a great deal of improvisation, invention, and experimentation and still carries with significant risk” (Selic, 2008). Major improvements in standardization and availability of tools must be made to further advance model driven engineering beyond academia. The authors therefore do not believe that for the time being a full-fledged model driven engineering approach in an industrial setting is feasible. This especially applies to the field of systems integration with particularities like one-off development, high heterogeneity, and multiple systems to be integrated. These and their implications on automated software development will be discussed in the following.

3. Characteristics of Systems Integration

Systems Integration comes with certain particularities, distinguishing it from conventional or single-system software development. It has to challenge a multiplicity of technologies, business processes, and other aspects, such as regulatory requirements. Considering the fact that most system integrators are active in multiple industries with multiple customers, chances that one project is similar to another are extremely small. However, the industrialization of software development requires some sort of specialization, standardization, and automation to be beneficial. While specialization can be found in Software Product Lines and standardization in Component Based Development, automation requires an approach similar to the ones introduced in chapter 2.

As with every new technology, implementation cost are associated with model driven engineering. First, one has to define a domain specific language in which the different applications of a product line will be modelled in. For systems integration, such a DSL needs to represent not only the system that is to be modelled, but also parts of those systems the new one is to be integrated with. Subsequently, respective model transformation engines and code generators must be developed, a task far from being trivial. Depending on the type of integration, such generators need to generate code for different platforms. Once all this is in place, automated software development may begin. Preparations therefore require a certain effort to be completed and must be considered from a cost benefit analysis. However, in the context of systems integration, implementation costs seem contradictory to model driven engineering. With the given situation and existing MDE concepts, it must be assumed that such intent will never break even, as no considerable economies of scale or scope exist to justify expenses for DSL, transformation engine and code generator, and IDE development. Furthermore, one has to consider shortcomings of current tools and development culture as introduced at the end of chapter 2. To overcome these challenges, either reusability or cost efficiency must significantly be increased, as well as suitable tools need to be available.

4. Combining MDE with Industrial Systems Integration

In our previous work (Minich *et al.*, 2010), we presented an organizational model for industrialized systems integration, which was done as a first step towards industrialization. It assumes that integration of different IT systems mostly occurs within the boundaries of a certain business domain, as the automotive industry, for instance. Herein, a large number of concepts, such as the logical entities car, supplier, or customer, remain the same for all applications and product lines. The model therefore consolidates similar activities of different product lines within a super ordinate layer, i.e. the Business Domain Layer. The advantage of this consolidation lies in a simplified integration of products from the underlying product lines, and a more efficient implementation approach due to the consolidation of redundant activities. In a subsequent step, we adapted the Business Component Model by Herzum and Sims (Herzum and Sims, 2000) as the second key principle of industrialization (Minich *et al.*, 2011). Herein we have shown how the different aspects of the model can be adapted to systems integration by matching them to an integration meta model. In addition we have shown where the required process steps of the Business Component Model are best situated within our Organizational Model for Industrialized Systems Integration.

This leaves us with automation as the final step, represented by model driven engineering. Given the MDE approaches introduced above, we chose Generative Programming as the basis for our work due to its focus on automating the development of a family member within a software product line (Czarnecki, 2005) and its ability to be implemented concurrently with the actual product being developed. Development within a product line allows for specialization as one of the key principles of industrialization. Advancing the approach while developing an actual product removes the necessity of high upfront investments. In the following sections we will show where in our previously developed organizational model the GP processes are best situated and how they relate to the Business Component Model.

4.1. Development Processes of Generative Programming

Generative programming (GP) includes the following eight main development processes (Czarnecki and Eisenecker, 2000) to define scope and functionality, infrastructure and core assets, as well as automation artefacts:

1. **Domain Scoping** identifies the domain of interest, stakeholders, goals, and defines the scope of the GP approach. It is influenced by e.g. the stability and maturity of potential solutions, available resources to implement them, and the potential for reuse during production (Czarnecki and Eisenecker, 2000).
2. **Feature & concept modelling** identifies the distinguishable characteristics of a system within a certain domain and models them within a feature model (Czarnecki and Eisenecker, 2000).
3. **Common architecture & component definition** depends on the previously developed feature model. Each identified area of functionality requires one or more components, whereas their component model, interaction, type, and distribution will depend on the architecture chosen for the system (Czarnecki and Eisenecker, 2000).
4. **Domain Specific Language design** specifies a language by defining its syntax and semantics. This may be done in different ways, ranging from simple translational semantics (i.e. defining a translation scheme to an implementation language) to complex axiomatic semantics (i.e. defining a mathematical theory for proving programs written in a given programming language) (Czarnecki and Eisenecker, 2000).
5. **Specification of configuration knowledge** defines how the problem space will be transformed into the solution space by utilizing the features and concepts identified above. It shields the developer from knowing all components and features by specifying illegal combinations, default settings, dependencies, or construction rules.
6. **Architecture & component implementation** implements the architecture and components identified above. The technology in which both are implemented depends on the scope of the domain.
7. **Domain Specific Language implementation** takes the DSL specification from the DSL design process and derives a concrete implementation. Here GP differentiates between separate DSLs (e.g. SQL or $\TeX$), embedded DSLs (e.g. template meta programming in C++), and modularly composable DSLs (e.g. embedded SQL, or aspect oriented programming) (Czarnecki and Eisenecker, 2000).
8. **Configuration knowledge implementation in generators** allows advancing the problem specified with the help of a Domain Specific Language into executable program code. To do so, generators apply validation of the input specification,

complete a given specification with default settings, perform optimizations, and eventually generate the implementation. Generators may be implemented as stand-alone programs, using built-in meta programming capabilities of a programming language, or by using a predefined generator infrastructure (Czarnecki and Eisenecker, 2000).

Comparing the eight process steps with the concepts of Software Product Line and Component Based Development, Generative Programming can be clearly subdivided into the industrial key concepts of specialization (steps 1 and 2), standardization (steps 3, 5 and 6), and automation (steps 4, 7, and 8).

4.2. GP and the Organizational Model for Industrialized Systems Integration

Software Product Lines and Component Based Development already cover the large parts of the GP processes. In the following we will therefore describe how our previously developed approach for Software Product Lines in systems integration needs to be adjusted to incorporate the requirements of Generative Programming.

4.2.1. The Business Domain Layer

The Business Domain Layer was developed to align domain wide functionality and utilize economies of scope due to similar concepts and core assets among different product lines of a given domain. It therefore contains the Software Product Line processes domain analysis & portfolio definition, architecture development & roadmap definition, and core asset development.

As to Generative Programming, the above processes already cover the GP processes 1 and 2, such as development of a domain or feature model (Minich *et al.*, 2010). Furthermore, the activities of GP processes 3 and 4 are already enclosed in Architecture Development & Roadmap Definition, and Core Asset Development. However, as the Business Domain Layer only features concepts suitable for more than one product line, we have to differentiate between global (business domain wide) and local (product line specific) aspects of GP. This means that there will for instance be DSL design activities in both, the Business Domain and the Software Product Line Layer. In the former, the overall structure and domain wide syntax and semantics are defined, whereas the latter covers product line specific syntax and semantics, such as “bill of materials” for a shop floor system produced in a particular software product line. The distribution is illustrated in Figure . Combining the activities introduced in (Minich *et al.*, 2010) with the respective ones from Generative Programming, the Business Domain Layer in its final stage consists of the following core processes:

- **Business Domain Analysis** explores the typical IT landscape of the business domain in scope and identifies areas of expertise required to develop and provide the products and services under consideration. Similar to software product lines but on a higher level, it identifies recurring problems and known solutions.
- **Portfolio Definition & Domain Scoping** evaluates the information from the domain model and develops a product portfolio for the particular business segment. The portfolio covers typical applications and solutions for the most

important business services of the segment and identifies the portfolio elements and resulting software product lines.
- **Architecture & Feature Definition**. Once the scope is defined, a basic product line and integration architecture, a component framework, and an overall feature model, applicable for all product lines are developed. As different product lines have different functional and technical requirements, this architecture may also exist in an abstract form and be instantiated within the product line subsequently. This approach allows for a later integration of products from different product lines of the same business domain.
- **Core Asset Development** develops reusable assets, applicable to all or many software product lines within the business segment. Such joint core assets may for instance be development tools and processes, or joint software development patterns. Core Asset Development may also include the production of reusable software components equal to each product line. To additionally support Generative Programming, Core Asset Development now also contains the definition of an abstract syntax for a domain wide specification language. This DSL may then be extended within the underlying software product lines in order to support more specific concepts.

4.2.2. The Software Product Line Layer

The Software Product Line Layer consists of several software product lines identified in business domain analysis and portfolio definition processes of the business domain layer (Minich *et al.*, 2010). The most obvious variance to a conventional software product line is the lack of the business domain analysis process, and a simplified domain requirements engineering process. These functions are now incorporated in the business domain layer and provide their findings to the subsequent product lines. All other processes remain the same but must adhere to the specifications and utilize the provided core assets from the business domain layer.

As to Generative Programming, we can find all but the first development process within the Software Product Line Layer. However, due to the separation of domain wide and product line specific concerns, the GP processes 2 to 4 only handle product line related concerns. A systems integrator's feature model for the automotive industry may for instance define the entity car with several features, such as model, engine, transmission, colour, price, owner, and so on. These features exist in all products of the underlying product lines. A product line for shop floor systems may however extend this feature model by adding features like electronic control unit (ECU) type, brake type, or parts list. As this has no implication on the functionality of the car itself or the customer, these features are not necessary to be known in other product lines. A financial system does not need to know what type of ECU is built into a car, but it does need to know the price and the owner of the car. This same principle applies to Common Architecture & Component Definition and Domain Specific Language Design. GP processes 5 to 5 are carried out in the software product lines only. Combining the activities introduced in our Organizational Model for Industrialized Systems Integration with the respective ones from Generative Programming, the Product Line Layer in its final stage consists of the following core processes:

- **Requirements Engineering & Feature Modelling** defines the scope of the intended software product line by identifying its products and documenting their commonalities and variability within a feature model. The process has to conform to the Portfolio Definition & Domain Scoping artefacts of the superior business domain layer, but may extend them with product line specific features.
- **Architecture, Component & DSL Design** transforms the scope defined in requirements engineering into a technical architecture and specification for the product line and its products. The architecture decomposes a software system into common and variable functional parts, and specifies the configuration knowledge in terms of component dependencies, default configurations, construction rules, illegal combinations, and rules for their implementation. Each identified area of functionality requires one or more components with an architecture specific component model, interaction scheme, and distribution mechanism. All programming artefacts are finally described within a Domain Specific Language. The process' activities must adhere to the specifications from the business domain layer, but may extend it with product line specific features.
- **Core Asset Development** provides the design and the implementation of reusable software assets (Pohl *et al.*, 2005). This implementation includes the overall framework, software components, executable code, and other product line assets, such as development processes and tools. In terms of Generative Programming, core asset development is also responsible for the implementation of the DSL as specified in the previous process. In a later and more mature stage, Core Asset Development will implement the configuration knowledge within generators to advance the system specified with the help of a DSL into intermediate models or executable code. As this can be extremely complex, we suggest postponing this activity until reasonable experience with the DSL and the product lines has been gathered.
- **Domain Testing** develops test cases and inspects all core assets and their interactions against the requirements and contexts defined by the product line architecture. Domain testing also includes validation of non-software core assets, such as business processes, product line architecture or development policies.
- **Software Integration** in the context of product line development occurs during pre-integration of several software components. They form blocks of functionality common to all products and contexts of a product line. Furthermore, the integration process ensures the interoperability of all reusable assets and provides the required integration mechanisms.

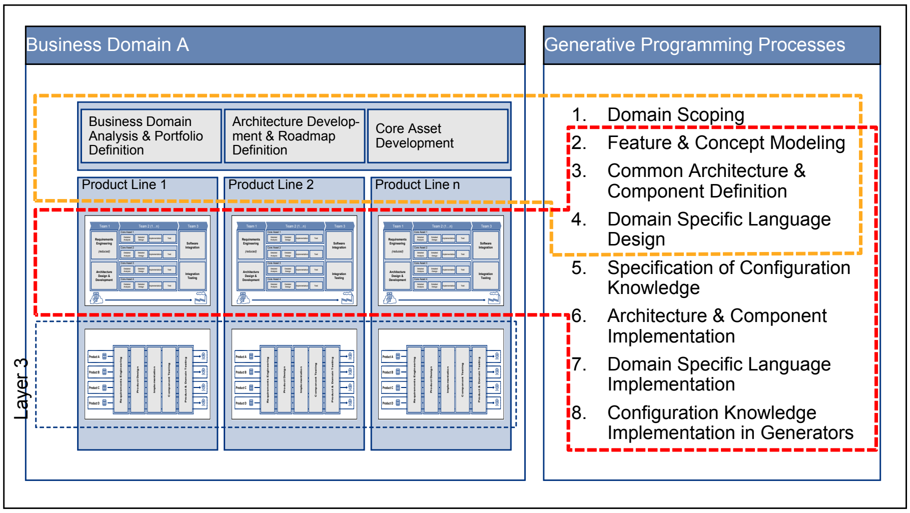
Business Domain A
Generative Programming Processes
Business Domain Analysis & Portfolio Definition
Architecture Development & Roadmap Definition
Core Asset Development
Product Line 1
Product Line 2
Product Line n
Layer 3
1. Domain Scoping
2. Feature & Concept Modeling
3. Common Architecture & Component Definition
4. Domain Specific Language Design
5. Specification of Configuration Knowledge
6. Architecture & Component Implementation
7. Domain Specific Language Implementation
8. Configuration Knowledge Implementation in Generators

Figure 1: Mapping of GP Processes to Organizational Structure

4.3. GP and the Business Component Model

The Business Component Model is a methodology to model, analyse, design, construct, validate, deploy, customize, and maintain large scale distributed systems, developed by Herzum and Sims (Herzum and Sims, 2000). It consists of five dimensions: Architectural Viewpoints, Component Granularity, Development Process, Distribution Tier, and Functional Categories. In our previous work we have already shown how to align the Business Component Model with our Organizational Model to reflect the particularities of Systems Integration. The following sections will show how these five dimensions fit together with Generative Programming, assuming that development of components occurs with GP.

4.3.1. Architectural Viewpoints

The first dimension consists of four architectural viewpoints, which are the Project Management Architecture (PMA, concerned with organizational decisions, tools, and guidelines), the Technical Architecture (TA, defining the execution environment, component and user interface frameworks, and other technical facilities), the Application Architecture (AA, describing development patterns, guidelines, or standards), as well as the Functional Architecture (FA, identifying the features and functional aspects of a system and their relationships).

With regards to the Project Management Architecture, Generative Programming does not make any statements about the organization or structure of a development project within its processes. The PMA from the Business Component Model is therefore regarded beneficial to the GP approach. In our organizational model, the PMA is found in the Business Domain Layer, whose organizational decisions, tools, and guidelines will influence the development in GP. The remaining three, rather technical viewpoints, are concerned with the execution infrastructure and programming frameworks (Technical Architecture), development patterns, guidelines, and programming standards (Application Architecture), as well as the functional aspects of a system including its implementation (Functional Architecture). Generative Programming in turn only offers the generic process common architecture & component definition. We therefore suggest replacing the respective GP process with the actual implementation of the much more detailed architectural viewpoints from the Business Component Model. For Generative Programming, this replacement offers a more comprehensive view on different aspects of the architecture, while for CBD it ensures coverage of more component related artefacts, such as the component infrastructure or execution environment.

4.3.2. Component Granularity

Generative Programming does not explicitly refer to well defined components as known from e.g. Enterprise Java Beans or Corba. Also it doesn't conceptually concentrate on business processes and therefore does not know reasonable levels of granularity. An artefact may for instance be a generic and reusable data container for C++, allowing handling domain specific types of information. It may also be a reusable programming library providing a complex business concept like a bank account. Generative Programming rather concentrates on technologies and means to develop reusable artefacts of variable sizes, depending on the intended usage. This way of partitioning a problem into reusable artefacts is known as *continuous recursion*. One iteratively partitions a problem into different but reasonable granularities. The Business Component model in turn follows a *discrete recursion* approach. It therefore defines five levels of granularity: the language class, the distributed component (a component in its common sense, e.g. an EJB or CORBA component), the business component (still independently deployable, consisting of distributed components and glue code, representing a business process), and the system level component (a set of business components providing business functionality). The highest level of granularity is the federation of system-level components (i.e. system level components federated to provide multiple complex business services).

We believe that discrete recursion and thus partitioning of the problem is more beneficial in an environment with systematic reuse. For each layer of recursion, a developer has to define scope, characteristics, packaging, and deployment (Herzum and Sims, 2000). In an environment where components are to be reused as much as possible, it seems more beneficial to define these layers of recursion on a common basis. A middleware messaging adaptor for a specific ERP system will most likely exist as a distributed component as introduced above. A developer can rely on this concept and build his application accordingly. We therefore suggest to introduce discrete recursion to the Generative Programming approach if it is to be used within component based development and systematic reuse in mind.

4.3.3. Development Process

The Business Component Model encompasses a set of manufacturing processes, which support component, system, and federation of systems development. However, as most organizations are in a transitive state towards CBD, Herzum and

Sims suggest a process called rapid system development (RSD). It is following the well known V-Model, whereas requirements to implementation denote the left, and component, system, and acceptance testing the right side of the V (Herzum and Sims, 2000). RSD allows subsequently engineering reusable artefacts based on customer specific requirements and eventually building the respective end product. The advantage is that reusable artefacts evolve on the fly. The disadvantage is that, beginning with the requirements of one specific customer, one may easily miss important variation points or even take architectural decisions which may conflict with the overall scope of the product line. Generative programming in turn focuses much more on domain engineering activities and the technical implementation of reusable artefacts, rather than development of the end product. It puts explicit focus on feature modelling processes such as FODA or FeatuRSEB (Czarnecki and Eisenecker, 2000), as all GP artefacts rely on a detailed domain model. As research in the field has progressed, we also considered PLUSS (Product Line Use Case Modelling for Systems and Software engineering) (Eriksson *et al.*, 2006) being a viable alternative for precise domain modelling. The advantage of PLUSS over FODA or FeatuRSEB is that besides a feature model it also allows to allocate use cases, use case variations, and cross-cutting concerns to each feature. In the context of the present work we follow the rationale of Generative Programming to define a precise model of the product domain before implementing any reusable artefacts. This seems especially important if domain specific languages and generators are to be built, although they will be rather simple in the beginning. We therefore suggest to enhance the Requirements, Analysis, and Design activities of Herzum and Sims' rapid system development process with Feature Modelling and Use Case Development of Eriksson et.al.'s PLUSS approach (Eriksson *et al.*, op. 2005). The result will be a detailed feature model, including a variety of use cases for the required feature combinations. Based on these artefacts, the customer specific application can be built and reusable components derived.

4.3.4. Distribution Tier

In their model, Herzum and Sims separate between user, workspace, enterprise, and resource tier. The user tier presents the component on the screen and communicates with the user. It may be stand-alone, plug in, or non-existent at all. The local business logic is implemented by the workspace tier, which will interact with the enterprise tier. Typical business logic may for instance include transaction management utilizing several enterprise-level resources. The latter are implemented by the enterprise tier, providing business rules, validation, and interaction between components. It typically forms the core functionality of business components of a complex, large-scale component based system. The resource tier manages access to shared resources, such as databases, files, or communication infrastructures and shields all higher layers from their technical implementation.

Such detailed differentiation of reusable components and their internal structure is not provided by the Generative Programming approach. Being more generic, GP leaves such decisions on the target architecture of the product line, which is in turn depending on the overall feature model (Czarnecki and Eisenecker, 2000). With regard to the Business Component model, feature model and architecture will already be available and are furthermore influenced by the conceptual structure of business components. In combination with GP, we see no issues when implementing the four distribution tiers with the means of Generative Programming.

4.3.5. Functional Categories

The final dimension defines utility, entity, process, and auxiliary business components (Herzum and Sims, 2000). Utility components can most generally be reused and represent autonomous concepts, such as unique number generators, currency converters, or an address book. Entity business components represent the logical entities on which a business process operates and are specific to a particular business domain. Examples are item, invoice, address, or customer. The actual business *process* is implemented within a process business component. Usually unique for one industry or customer, it is hardly reusable. The fourth category, auxiliary business components, provides services usually not found within a process description. Such may be performance monitoring, messaging, or middleware services.

As with the distribution tier above, Generative Programming does not know any functional categories. However, a detailed feature model in connection with component granularity, distribution tiers, and functional categories, will provide a structured and standardized approach to generative development of business components. As such we believe it is more likely to yield systematic reuse than a structure that is flexible from component to component.

5. Conclusion & Further Research

As we have explained in chapter 3, systems integration comes with certain particularities requiring a highly efficient and cost effective way of implementing industrial key concepts. The low number of similar products in SI seems contradictive to Model Driven Engineering with its Domain Specific Languages, Model Transformers, and Code Generators. However, with integrating GP into our organizational model and combining it with CBD, we can save efforts for domain scoping, feature and concept modelling, architecture and component definition, configuration knowledge specification, and component implementation. All these activities, although slightly adapted, have already been completed, once it comes to the implementation of MDE.

Together with the Business Component Model, a standardized component and implementation architecture is available which allows us to systematically reuse functionality already developed. If a middleware adaptor will always be implemented as an auxiliary business component at the resource distribution tier, it is much more likely to be reused than a freely implemented one. We therefore believe that in order to get the most out of Generative Programming, it must be combined with a component based development approach. Based on our previous work (Minich *et al.*, 2011), we found the Business Component Model to be most beneficial, especially in the context of systems integration.

The present paper completes the development of a concept for industrialized systems integration. It consists of the organizational model for software product lines in systems integration, reflecting specialization as the first industrial key principle. Subsequently, the alignment of Herzum and Sims' Business Component Model with Vogler's Integration Meta Model describes how to divide a system into a set of reusable artefacts, reflecting the particularities of systems integration. With the present work, Generative Programming has been identified as a potential way towards automation as the final industrial key principle. What is left to be done is a concluding description of the overall concept including the presentation of a field study across all three principles: Beginning with Business Domain design and subsequent Software Product Line definition, over the development of a detailed feature model and component structure, up to the definition and implementation of an initial Domain Specific Language and the according generators with the help of Generative Programming. It is intended to exemplarily develop at least one example of each artefact required for a successful industrialization of systems integration.

6. References

Clements, P. and Northrop, L. (2007), *Software product lines: Practices and patterns,* [Nachdr.], Addison-Wesley, Boston.

Czarnecki, K. (2005), "Overview of Generative Software Development", in Banâtre, J.-P., Fradet, P., Giavitto, J.-L. and Michel, O. (Eds.), *Unconventional Programming Paradigms*, *Lecture Notes in Computer Science*, Vol. 3566, Springer Berlin / Heidelberg, pp. 97-97.

Czarnecki, K. and Eisenecker, U. (2000), *Generative programming: Methods, tools, and applications*, Addison Wesley, Boston.

Encyclopaedia Britannica (2005), "Industrial Revolution", in *Encyclopaedia Britannica: In 32 volumes,* Vol. 6, 15. ed., Encyclopaedia Britannica, Chicago, London, New Delhi, Paris, Seoul, Sydney, Taipei, Tokyo, pp. 304–305.

Eriksson, M., Börstler, J. and Borg, K. (op. 2005), "The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations", in Obbink, H. and Pohl, K. (Eds.), *Software product lines: 9th international conference, SPLC 2005, Rennes, France, September 26-29, 2005 proceedings*, Springer, Berlin, New York, NY, pp. 33–44.

Eriksson, M., Börstler, J. and Borg, K. (2006), "Software Product Line Modeling Made Practical", *Communications of the ACM*, Vol. 49 No. 12, pp. 49–53.

Fischer, J. (1999), *Informationswirtschaft Anwendungsmanagement*, Oldenbourg, München, Wien.

Greenfield, J., Short, K. and Cook, S. (2004), *Software factories: Assembling applications with patterns, models, frameworks, and tools*, Wiley, Indianapolis, Ind.

Hasselbring, W. (2000), "Information System Integration", *Communications of the ACM*, Vol. 43 No. 6, pp. 32–38.

Herzum, P. and Sims, O. (2000), *Business component factory: A comprehensive overview of component-based development for the enterprise*, John Wiley, New York.

Leser, U. and Naumann, F. (2007), *Informationsintegration: Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen,* 1. Aufl., dpunkt-Verl., Heidelberg.

Minich, M., Harriehausen-Mühlbauer, B. and Wentzel, C. (2010), "An Organizational Approach for Industrialized Systems Integration".

Minich, M., Harriehausen-Mühlbauer, B. and Wentzel, C. (2011), "Component Based Development in Systems Integration", in *GI Lecture Notes in Informatics 2011: Informatik schafft Communities*, Ges. für Informatik, Bonn, p. 470.

Object Management Group (2003), *MDA Guide Version 1.0.1*.

Petrasch, R. and Meimberg, O. (2006), *Model Driven Architecture: Eine praxisorientierte Einführung in die MDA,* 1. Aufl., dpunkt, Heidelberg.

Pohl, K., Böckle, G. and Linden, F. (2005), *Software product line engineering: Foundations, principles, and techniques ; with 10 tables*, Springer, Berlin.

Riehm, R. (1997), "Integration von heterogenen Applikationen", Dissertation, Universität St. Gallen, St. Gallen, 1997.

Selic, B. (2008), "Personal reflections on automation, programming culture, and model-based software engineering", *Automated Software Engineering*, Vol. 15 3-4, pp. 379-391.

Singh, Y. and Sood, M. (2009), "Model Driven Architecture: A Perspective", in IEEE Computer Society (Ed.), *Proceedings of the 2009 IEEE International Advance Computing Conference*, IEEE Computer Society, Patiala, pp. 1644–1652.

Stahl, T. and Bettin, J. (2007), *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management,* 2., aktualisierte und erw. Aufl., dpunkt-Verl., Heidelberg.