# Variability Mechanisms in E-Business Process Families*

Arnd Schnieders and Frank Puhlmann

Business Process Technology Group
Hasso-Plattner-Institute for IT Systems Engineering
at the University of Potsdam
D-14482 Potsdam, Germany
{arnd.schnieders, frank.puhlmann}@hpi.uni-potsdam.de

**Abstract** Nowadays, process oriented software systems, like many business information systems, don't exist only in one single version, but in many variants for better coverage of the target market. Until now, the corresponding customization has to be done manually, which is a time-consuming and error-prone task, which could be realized much more efficiently by applying process family engineering techniques. Process family engineering is a modern software development approach, which allows for the rapid and cost-effective development and deployment of customer tailored process oriented systems. In this paper we present our findings in the area of process family architectures for e-business systems, described as variant-rich process models in the Business Process Modeling Notation. We moreover address variability implementation issues using Java variability mechanisms and code generators.

## 1 Introduction

Nowadays, process oriented software systems, like many business information systems, don't exist only in one single version, which covers the whole target market. Instead, many different variants of the system exist, which are specialized according to diverging customer needs. Until now, the corresponding customization has to be done manually, which is a time-consuming and error-prone task. However, the ability to rapidly and cost-effectively develop and deploy customer tailored system variants is crucial to the competitiveness of a company developing business software. In order to cope with these challenges, techniques for the efficient production of similar software systems have been developed. These techniques, known as software product family engineering [1], have already been applied successfully in many enterprises [2]. However, up to now the investigation of product family engineering techniques for families of process oriented software, in short process family engineering, has been widely neglected, which leads to an urgent need for research in this area. In this paper we therefore present our findings in the area of process family architectures for e-business

* The work reported in this paper has been supported by the German Ministry of Research and Education by the PESOA project

systems, described as variant-rich process models in the Business Process Modeling Notation (BPMN) [3] as well as variability implementation issues using implementing variability mechanisms and code generators.

This paper is structured as follows: In section 2 we give a brief introduction to some basic concepts and describe in section 3 their application to an e-business process family. Section 4 defines a set of variability mechanisms for process family architectures and examples for their implementation in Java. In section 5 we illustrate our findings based on an exemplary process family of e-business shops. In section 7 we summarize the contents of this paper and give an overview of related work.

## 2 Preliminaries

In this section we give a brief introduction to Process Family Engineering, Process Family Architectures, and Generative Programming.

### 2.1 Process Family Engineering

Product family engineering is a paradigm to develop software applications using a set of software subsystems and interfaces that form a common structure based on which derivative products tailored to individual customer needs can be efficiently developed according to [4]. Another important aspect is that within a software product family reuse isn't restricted to the reuse of implementation artifacts but is expanded to any development artifact (like e.g. requirement or design models).

Product family engineering is characterized by a so called dual lifecycle [5] as indicated in figure 1 [6]. In order to emphasize that our work focuses on the development of process-oriented software, we use the term process family engineering instead of product family engineering and process family infrastructure instead of product family infrastructure. However, the basic development process is the same for product family engineering as for process family engineering. In the first section of the process family development process (called process family engineering) generic development artifacts (called the process family infrastructure) are developed based on which process family members are derived efficiently in the corresponding phase within the second section (called application engineering) of the process family engineering process.

### 2.2 Process Family Architectures

During the design of a process family a process family architecture (PFA) is developed based on the process family requirements. The PFA acts as a reference architecture for the members of the process family and describes the basic structure for the applications of the process family. It defines the reusable system parts with their interfaces and covers both, the functional as well as the non-functional requirements on the process family. Moreover, the PFA describes

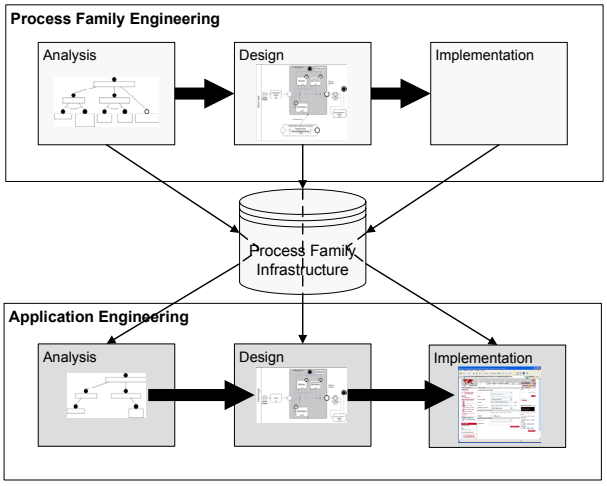



**Figure 1.** Process Family Engineering Process

which techniques shall be applied for realizing the variability (i.e. the variability mechanisms) and on which variation points they shall be applied. The selection of appropriate variability mechanisms is crucial for the design of the process family since they can have a substantial impact on the functional and non-functional properties of the system. Additionally, the proper selection of a variability mechanism guarantees for an easy generation of process family members based on the process family infrastructure.

Thus, for supporting process family engineering, concepts and a notation for process family architecture variability mechanisms (PFA variability mechanisms) are required, which allow for modeling architecturally relevant decisions concerning the realization of the system's variability. Figure 2 describes the dependencies between the process family requirements, the process family architecture, PFA variability mechanisms and implementing variability mechanisms. The model is structured according to the three phases of process family engineering into three packages: Analysis, Design and Implementation. The requirements on the process family members are realized by a corresponding PFA. The variability in the process family is modeled by means of variation points to which variants can be bound by means of PFA variability mechanisms. The variability mechanisms represented in the PFA are realized in the program code by so called implementing variability mechanisms. During the implementation different variability mechanisms can come into question, which can show different binding times. Which variability mechanisms are available highly depends on the application domain and the system to be implemented. The idea is that a set of implementing variability mechanisms with the same functional properties shall be represented by the same PFA variability mechanisms. Thereby, the variability modeling takes into account the functional and the resulting non-functional properties of the

variability implementation. Moreover, a binding time can be specified for the variability, which is also application domain dependent.

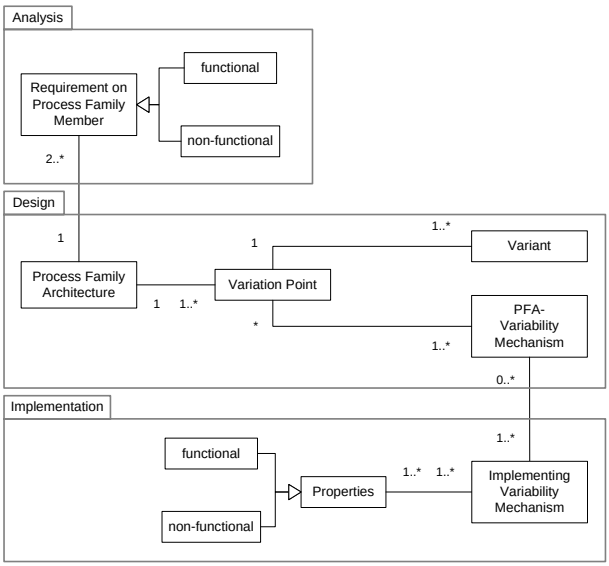



**Figure 2.** Role of Variability Mechanisms in Process Family Engineering

### 2.3 Generative Programming

According to [7] "Generative Programming (GP) is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge". So, in short, generative programming can be used to generate members of a product (and therefore also a process) family. For specifying a member of the process family configuration models are used. The configuration model based specification of a process family member is then processed by the program generator to generate the desired process family member.

## 3 E-Business Process Family

In this section we describe how the concepts described in section 2 apply to the development of an e-business process family. We thereby follow the process family engineering reference model shown in figure 1.

Concerning the development of the process family infrastructure we use the extended feature diagram notation suggested by [7] to describe the requirements on the process family during the analysis phase. Based on these requirements a process family architecture is developed during the design phase using BPMN enhanced by the PFA variability mechanisms for variability modeling as described conceptually in section 2.2 and concretely in the following sections. We assume, that dependencies between variabilities spreaded over the process family architecture can always be traced back to the selected feature configuration. Thus, any variability in the process family architecture has to be linked to features or feature combinations in order to be resolvable properly during application engineering. However, since these linking descriptions can become arbitrarily complex, we will show only a simplified version in the process model (see section 4.2), while the full linking information has to be handled by a tool. In the implementation phase a program generator for the process family is implemented based on the process family architecture. Here we assume the generator to employ the frame technology for software generation developed by [8], which is based on the frame/slot concept for knowledge representation invented by [9]. Concerning the variability implementation, since we assume the e-business application to be written in Java, Java specific variability mechanisms are used to implement the PFA variability mechanisms applied in the BPMN based process family architecture. However, our approach is not restricted to Java but can be applied to any programming language and the programming language specific variability mechanisms. In the analysis phase of application engineering the requirements on the e-business system to be developed are described by deriving an application-specific feature model from the feature model created during process family infrastructure development. Based on this feature model an application specific process architecture (i.e. a BPMN model) is derived from the process family architecture model (i.e. the variant rich BPMN model). The application specific BPMN model is used as configuration model for the program generator. Based on the BPMN model the generator derives application specific Java components from the variant rich Java components and assembles them to the desired e-business system.

## 4 Variability Mechanisms for E-Business Processes Families

This section introduces a set of architecturally relevant variability mechanisms for e-business process families. First we will give an overview of different categories of variability mechanisms in section 4.1 and a description of the respective variability mechanisms in subsection 4.2.

### 4.1 Categories of Variability Mechanisms

In general, variability mechanisms can be categorized into basic variability mechanisms and variability mechanisms, which are derived from the basic variability

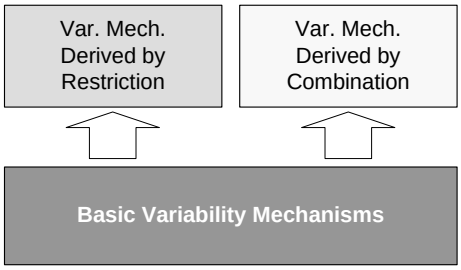



**Figure 3.** Categories of Variability Mechanisms

mechanisms. As the name indicates, basic variability mechanisms are standalone mechanisms, which don't require any other variability mechanisms. We have identified four types of basic variability mechanisms: *encapsulation of varying sub-processes*, *parameterization*, *addition/omission/replacement of single elements*, and *data type variability*. Since data flow is not supported directly in BPMN, we will leave out the variability mechanism *data type variability*. Moreover, we support *addition/omission/replacement of single elements* only in a restricted form by *BPMN inheritance*, which we will introduce in the following section.

Concerning the second category of derived variability mechanisms we can further divide this category into variability mechanisms derived by restriction and by combination of other variability mechanisms as figure 3 shows. With *BPMN inheritance* and *extension* we introduce two examples for variability mechanisms derived by restriction and with *design patterns* an example for a variability mechanism derived by combination.

### 4.2 Variability Mechanisms

Each mechanism is described in four parts. A short description of the functionality is followed by a BPMN representation. Thereafter, implementing variability mechanisms in Java are discussed. While Java variability mechanisms are used for variability implementation a code generator employing the frame technology for software generation is used for configuration. We give a short example for variability mechanism implementation and configuration in section 5.1. For every variability mechanism we give continuative references.

According to the requirements for a process family architecture stated in section 2.2, a variant-rich business process diagram needs to contain three additions to standard business process diagrams. The first addition is a marking of the places where variability occurs (variation point). Second, the possible resolutions (variants) should be shown in the diagram. Third, the variability mechanism used to derive the resolution should be shown.

Regarding the first requirement, the identification of variation points, we propose to adapt the concept of a stereotype from the UML2 specification to BPMN. Each activity, association, and artifact can have a stereotype attached.

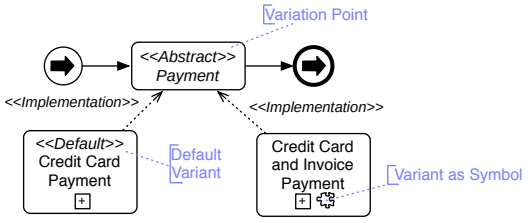



**Figure 4.** Encapsulation in BPMN.

For the purposes of a variant-rich business process diagram, the introduction of a stereotype called ≪*VarPoint*≫ to represent a variation point and ≪*Variant*≫ to represent a variant are sufficient. The ≪*Variant*≫ stereotype can also be expressed graphically as a puzzle-piece like marker at the bottom of an activity. For a more convenient illustration, a third stereotype ≪*Variable*≫ can be used to denote variability below the level of detail currently shown.

The stereotype ≪*VarPoint*≫ can be further specialized. An ≪*Abstract*≫ variation point represents alternative behavior; it has to be resolved with a specific variant. A ≪*Null*≫ variation point represents optional behavior; it can be resolved with a specific variant. ≪*Alternative*≫ is a short representation of an abstract variation point with a specific variant which is the default resolution to this variation point. An ≪*Optional*≫ variation point is a short representation of a ≪*Null*≫ variation point and a specific variant. The stereotype ≪*Variant*≫ can have the tagged value *feature* which provides information about the dependency of the subprocess variant from a certain feature configuration. Since these dependencies can become arbitrarily complex, we provide only a simplified version in the process family architecture. If the selection of a subprocess variant depends on the selection of a number of features, the *feature* tagged value will contain these features as a list of comma-separated feature names. However, if the dependency is more complex, e.g. if a certain subprocess variant shall only be selected if feature 1, 2, and 3 have been selected, but not feature 5 or 6, the *feature* tagged value would contain the following expression: *F(feature 1, feature 2, feature 3, feature 5, feature 6)*. This indicates that the dependeny is a more complex one depending on the selection of the features given as parameter values. The concrete dependency information has to be maintained in a different document, or by a tool.

### 4.3 Basic Variability Mechanisms

#### Encapsulation of Varying Sub-Processes

*Functionality.* Application-specific subsystem implementations are inserted into an invariant subsystem interface.

| Implementing Variability Mechanism (Java) | Slot configuration | Binding Time |
|---|---|---|
| polymorphism with subclasses | insertion of variant specific subclass invocation/object generation statement | compile time<br>runtime |
| interface implementation | insertion of variant specific method body into method header | compile time |
| dynamic class loading | insertion of variant specific class loading and subsequent typecast statement | runtime |
| static libraries | integration of variant specific libraries during compilation | link time |

**Figure 5.** Implementing Variability Mechanisms for PFA Variability Mechanism Encapsulation of Varying Sub-Processes

*PFA Variability Mechanism.* A BPMN sub-process can hide alternative sub-processes behind an invariant interface. Thereby, an interface is defined as the set of input and output events of an activity. The interface activity is marked with the stereotype ≪*Abstract*≫. Possible realizations of the interface are connected using associations marked with ≪*Implementation*≫. Figure 4 shows the encapsulation of a sub-process. The alternative behavior can occur at a task marked with ≪*Abstract*≫ and the name of the variation point, which is *Payment* in the figure. Possible implementations are shown as separate sub-processes, either collapsed or expanded. If there exists a default implementation, it can be marked with ≪*Default*≫, like the sub-process *Credit Card Payment* in the figure. A directed association from the implementation sub-process to the variation point marks the sub-process as a possible resolution to the variation point. The associations have to be marked with the stereotype ≪*Implementation*≫. Note the use of a graphical symbol to represent the stereotype ≪*Variant*≫ at the bottom of the sub-process *Credit Card and Invoice Payment*.

*Implementing Variability Mechanism.* We assume that BPMN activities are represented in Java as methods and BPMN pools as Java components providing an interface of method declarations. The invariant Java method declarations encapsulate varying method implementations. Java provides several variability mechanisms that support variation in Java method implementations as shown in the left column of figure 5. The respective configuration work performed by the code generator for deriving a product variant is indicated in the middle column, while the binding time for the implementing variability mechanism is show in the right column. We thereby follow the binding time model in [10]. The implementing variability mechanisms utilized here can be found in [10], and [11].

*References.* Encapsulation is a variability mechanism also pointed out by [12, 13] to be relevant on a design model level. Encapsulation is also referred to by [14] as the utilization of black box components.

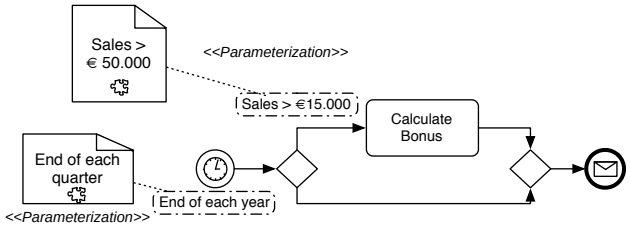



**Figure 6.** Parameterization in BPMN.

| Implementing Variability Mechanism (Java) | Slot configuration | Binding Time |
|---|---|---|
| configuration files + java branches | setting of variant specific configuration file entry | runtime |
| database entries + java branches | setting of variant specific database entry | runtime |

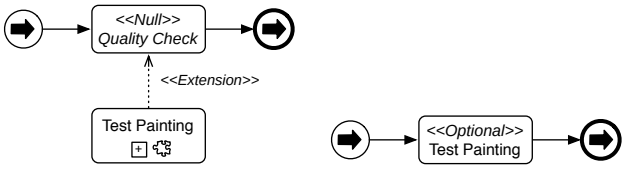
**Figure 7.** Implementing Variability Mechanisms for PFA Variability Mechanism Parameterization

### Parameterization

*Functionality.* Using parameterization variants of subsystems are generated by configuring a generic subsystem with a set of parameter values. The prerequisite for parameterization is that all possible variants are provided in the subsystem's code.

*PFA Variability Mechanism.* In BPMN, each attribute can be parameterized to support optional, alternative, or range variation points. Range variation points denote enumerations like $\{Mo, We, Su\}$ or a range of continues values as $0 \cdots 300$. For a graphical representation, the attribute is written beside the element and surrounded with a grouping box. Associations are used to link variant data objects that contain the possible parameters to the grouping box that surrounds the attribute. The association is marked with the stereotype $\ll Parameterization \gg$. Figure 6 shows the parameterization of two different attributes. The upper one parameterizes the ConditionExpression attribute of a sequence flow. The default value is a guard that activates the sequence flow if the sales are greater then €15.000. An alternative parameterization changes the attribute to activate the sequence flow if the sales are greater then €50.000. The lower one offers an alternative for the TimeDate attribute of the intermediate timer event. The default behavior triggers the event at the end of each year, whereas the alternative behavior triggers the event at the end of each quarter.

*Implementing Variability Mechanism.* Figure 7 gives two examples for implementing parameterization in Java using a code generator for configuration.




(a) Null Activities. (b) Optional Activities.

**Figure 8.** Extension in BPMN.

*References.* Parameterization is referenced as variability mechanism by [15, 12, 16].

### 4.4 Variability Mechanisms Derived by Restriction

#### Extension

*Functionality.* Extensions and extension points are used to extend an encapsulated subsystem at predefined points, the extension points, by additional optional behavior selected from a set of possible variants.

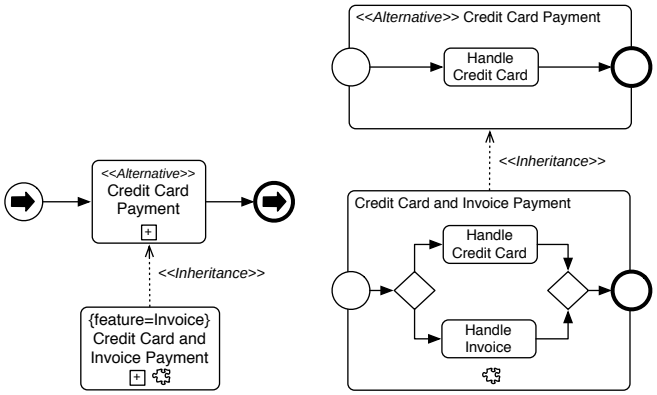
*PFA Variability Mechanism.* Extension points use a combination of encapsulation and null sub-processes to realize optional variation points. An extension point activity is marked with the stereotype ≪*Null*≫. Associations marked with ≪*Extension*≫ connect optional implementations. Figure 8(a) uses extension points to realize optional behavior. The extension point *Quality Check* is marked with the ≪*Null*≫ stereotype. Possible resolutions are attached with associations labeled with an ≪*Extension*≫ stereotype. If there is only one optional resolution of a variation point, it can be marked with the stereotype ≪*Optional*≫ and directly placed between the sequence flows, without the use of a ≪*Null*≫ task, shown in figure 8(b).

*Implementing Variability Mechanism.* For realizing extensions in Java implementing variability mechanisms for encapsulating the extending and varying sub-process implementations are required, which have been discussed in section 4.3. Moreover, a Null-Activity implementation is required, which doesn't perform any calculation but sticks to the interface of the Null-Activity (also including any possible exceptions that might occur during the execution of the activity).

*References.* Extensions/Extension Points are a very common variability mechanism referred to in many publications [17–19, 16].

#### Inheritance

*Functionality.* Inheritance adds restrictions to addition/omission/replacement of single elements. Special inheritance transformation rules preserve the structural integrity of the processes.




(a) Simple Task Inheritance. (b) Sub-Process Inheritance.

**Figure 9.** Inheritance in BPMN.

*PFA Variability Mechanism.* Inheritance modifies an existing (default) sub-process by adding activities and pools regarding to specific rules. This allows for realizing alternative variation points. An association represents inheritance from the child activity to the parent activity when it is marked with the stereotype ≪*Inheritance*≫. Figure 9(a) shows alternative behavior by the use of inheritance. The default variant is shown at the top of the figure, placed between the sequence flows. It is marked with the ≪*Alternative*≫ stereotype as it is a special notation of alternative behavior by encapsulation. The other alternative is realized by inheritance, which is indicated by the ≪*Inheritance*≫ stereotype at the association between the two sub-processes. The specialized sub-process *Credit Card and Invoice Payment* belongs to the feature invoice as annotated with the tagged value feature. The stereotype ≪*Variant*≫ is shown as a graphical marker (the puzzle piece like symbol at the bottom of the sub-process). Figure 9(b) shows the expanded sub-processes of figure 9(a). It can be seen that the task *Handle Credit Card* is reused in the specialization *Credit Card and Invoice Payment*.

*Implementing Variability Mechanism.* As shown in figure 10 BPMN inheritance transformations comprising the addition and replacement of activities and pools can be realized using Java „conditional compilation“ or by solely using frames/slots.

*References.* Inheritance is referred to by [17, 12, 19, 20] as a variability mechanism on the model level as well as on the code level.

| Implementing Variability Mechanism (Java) | Slot configuration | Binding Time |
|---|---|---|
| "java conditional compilation" | setting of variant specific value for static variable | compile time |
| pure frame/slot concept | optional or variant specific inclusion of method invocation (local or remote method invocation) | compile time |

**Figure 10.** Implementing Variability Mechanisms for PFA Variability Mechanism Inheritance

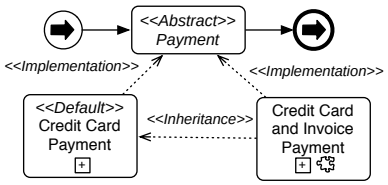



**Figure 11.** Strategy Pattern in BPMN.

### 4.5 Variability Mechanisms Derived by Combination

#### Design Patterns

*Functionality.* Here we will concentrate on the Strategy Pattern as one of the design patterns referenced most frequently in the context of product family engineering. The idea of the strategy pattern is to make different algorithm variants, which are hidden behind a common encapsulated interface, interchangeable. The algorithm variants are derived from a default algorithm variant using inheritance. Thus, the pattern combines encapsulation and addition/omission/replacement of single elements.

*PFA Variability Mechanism.* The concepts of encapsulation and inheritance can be used to implement design patterns that describe variability. There are no additional graphical notations required; the patterns can be formed by the use of the above mentioned concepts. Figure 11 implements the strategy design pattern. It is derived from figure 4 with an additional inheritance relation between *Credit Card and Invoice Payment* and *Credit Card Payment*.

*Implementing Variability Mechanism.* Since the strategy pattern is a combination of the two variability mechanisms encapsulation of varying sub-processes and inheritance, a combination of the respective implementing variability mechanisms can be used for implementation.

*References.* Design Patterns [21] like 'Adapter', 'Strategy', 'Template Method', 'Factory', 'Abstract Factory', 'Builder', and 'Decorator' [22] Pattern are frequently referred to as variability mechanisms [17, 16, 12]. However, according to

[23], except for a small number of Design Patterns any Design Pattern provides a way to implement variability.

## 5 Example

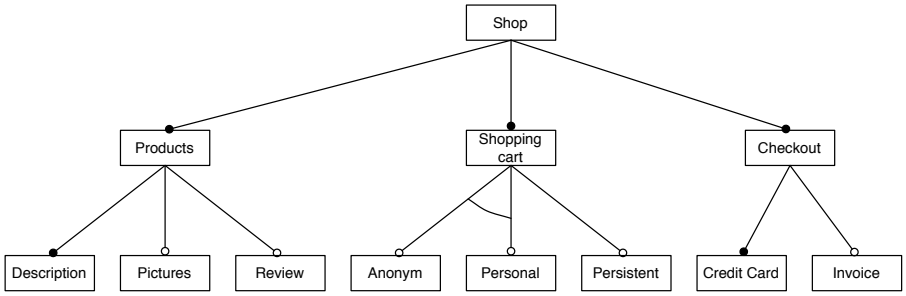



**Figure 12.** Features of the E-Business Shop.

This section presents a variant-rich process model for a process family of e-business shops also demonstrating some variability implementation details. A feature model is given in figure 12. An e-business shop consists of three mandatory features: products, shopping carts, and checkout. A product has a description as mandatory sub-feature, as well as optionally pictures and reviews. A shopping cart has three optional sub-features. It can either be anonymous (e.g. accessed by a proxy) or personalized. A personalized shopping cart allows for customer dependent discounts. Furthermore, a shopping cart can be made persistent, meaning that each time the customer returns, the shopping cart is loaded. The checkout has one mandatory sub-feature, offering a credit card checkout and an optional sub-feature invoice checkout.

Figure 13 contains the variant-rich high-level process model of the e-business shop process family. We omitted the feature anonymous shopping cart for the sake of simplicity. A new instance of the shop's workflow is triggered by the customer starting a new browser instance, thereby also creating a new instance of the shop's process (denoted with $\ll New \gg$). The customer then explores and choses products in interaction with the shop, where the shop delivers product information and composes the shopping cart. If the customer decides to buy, she triggers the checkout sub-process of the shop. The shop then sends the delivery and both parties additionally handle invoice payment if the feature is included in this particular shop configuration.

The variant elements have been realized by different variability mechanisms. The customer's pool contains the optional behavior *InvoiceCustomer*, which is represented by a null activity. If the feature *Invoice* is selected, the sub-process *Customer Invoice Payment* is included at the extension point. The shop's pool

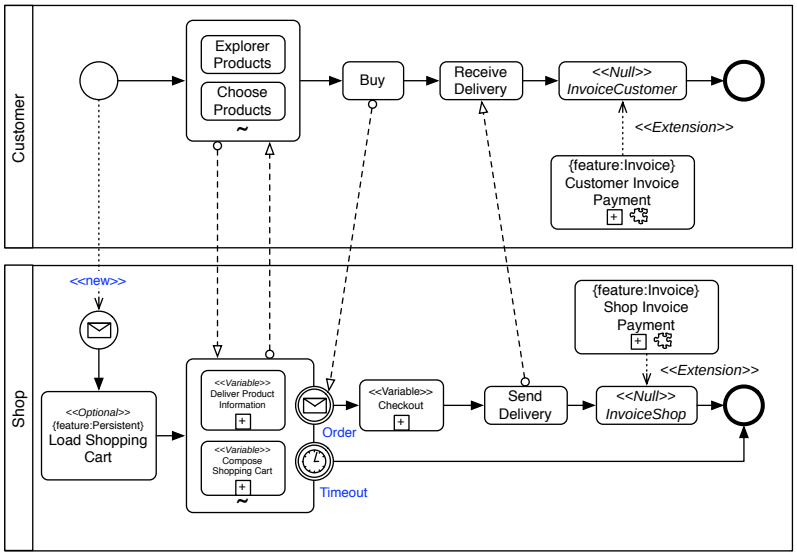



**Figure 13.** Top-level Process Model of the E-Business Shop.

has the optional task *Load Shopping Cart*, which is included if the feature *Persistent* (shopping cart) is selected. The null activity *InvoiceShop* is filled with *Shop Invoice Payment* if the invoice feature is selected. The task *Shop Invoice Payment* corresponds to the *Customer Invoice Payment*. As both variation points are enabled by the same feature, their realizations always appear together. The sub-processes *Deliver Product Information*, *Compose Shopping Cart*, and *Checkout* contain variability at a lower level, denoted with the «*Variable*» stereotype. For a lack of space we only consider the *Checkout* sub-process in detail.

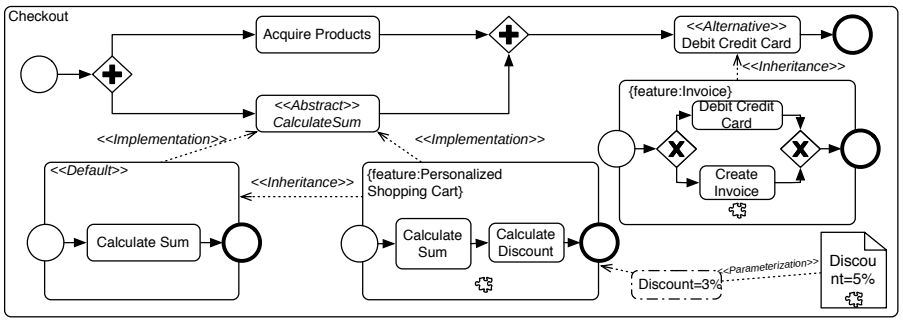



**Figure 14.** Checkout Sub-Process of the E-Business Shop.

Figure 14 expands the sub-process *Checkout*. It uses the concept of design patterns to describe the possible resolutions to the alternative variation point *CalculateSum*. The first resolution implements the default behavior, i.e. it only calculates the sum. The second, alternative resolution, specializes the default one by using inheritance to add the additional calculation of a discount. The percentage of the discount is parameterized with a default value of 3. The task *Debit Credit Card* also has an alternative implementation derived by using inheritance.

### 5.1 Implementation

In this section we will give a short example of how to implement the variability within a variant-rich BPMN process model using Java specific implementing variability mechanisms and a frame based code generator for configuration. Let's assume class *CalculateSum* in figure 15 implements the respective BPMN activity in figure 14. The common interface for the two *CalculateSum* resolutions is represented by the method interface *calculateSumSubprocess*. In order to implement *BPMN inheritance* the imlementing variability mechanism *Java conditional compilation* is used, which is realized using the static final boolean variable *PERSONALIZED_SHOPPING_CART*. If the variable value is set to *true* the shop feature *Personal* will be regarded in the target code. Else, the respective code lines, which depend on the evaluation of the if-statement in line 90 will not be considered by the Java compiler. For setting the variant specific value of the *PERSONALIZED_SHOPPING_CART* variable, a *slot* has been defined, which can be resolved during application engineering by selecting the corresponding resolutions. Figure 15 also implements the configuration of the activity *CalculateSum*. Therefore, a parameter value is read from a configuration file leading to a parameter dependent processing (line 240). The two alternative method invocations (in line 280 and 330) shall illustrate that arbitrary parameter value dependent processing could be performed. The respective configuration point for the code generator is the parameter value in the configuration file, which isn't shown in the example.

## 6 Acknowledgements

The findings presented in this paper have been validated in a case study of an e-business process family as part of the research project PESOA [24]. This case study has been carried out together with a Bachelor project (with the name „Magrathea“) at the Hasso-Plattner-Institute in cooperation with our PESOA industry partners ehotel AG and Delta Software Technology GmbH. Ehotel provided the e-business processes, while Delta Software Technology provided the tool HyperSenses [25] for code generator implementation and configuration. We want to seize the opportunity to express our sincere gratitude to Cord Giese and Winfried Buhl of Delta Software Technology as well as to Dr. Matthias Kose of ehotel AG for their nice cooperation. We also want to give our special thanks to the students of the „Magrathea“ project for their enthusiasm in performing the case study.

## 7 Conclusions

Process Family Engineering hasn't been considered adequatly in research so far. Therefore, in this paper we have introduced an approach for process family architecture modeling and implementation, which contributes to a rapid and cost-effective development and deployment of customer tailored process-oriented business information system variants.

We have shown how a process family architecture for a family of e-business systems can be modeled in BPMN. For representing variability in BPMN based process family architectures we have introduced a set of variability mechanisms for BPMN and outlined their implementation as well as their configuration using HyperSenses program generators.

For modeling product family architectures a number of alternative techniques have been proposed. There are for example approaches following the separation of concerns principle. Examples are the Hyper/UML approach [26] and two investigations on the composition of Statechart diagrams [27, 28]. Another group of publications suggests the explicit expression of variability within reference architectures by means of UML annotations [29, 30]. Non-graphical approaches comprise MOF-based product family architectures, which can be represented as XMI files [31] and the parameterization of BPEL [32] processes [33]. Another publication aims at representing EPC [34] based reference models [35]. Finally, a group of techniques could be referred to as template approaches [36, 12]. The work presented in this paper can be classified as a template approach. However, in contrast to existing template approaches with our variability mechanism centric approach we want to allow for making decisions concerning the realization of the product family variability during product family architecture development. These decisions and their visualization shall then enable the requirements-driven development of product family implementation artifacts and the easy derivation of application-specific artifacts. Thereby, we focus on the behavioral aspects of process-oriented systems, which have been neglected in product family engineering research so far.

## References

1. Becker, M.: Adaptation Support in Software Product Families (in German). PhD thesis, Technical University of Kaiserslautern (2004)
2. Reuys, A., Pohl, K., Gacek, C., Bermejo, J., Martínez, J.M., van der Sterren, W., Känsälä, K., Vehkomäki, T., Lerchundi, R., Martínez, R.A.C., Dueñas, J.C., Mittrach, S., Waeber, F., Berde, B., Sophie, V.: System Family Process Frameworks. ESAPS deliverable ESI-WP2-0002-04, University of Essen, Fraunhofer IESE, Sainco, Philips, Nokia, European Software Institute, Universidad Politécnica de Madrid, Siemens, Thomson-CSF/Alcatel LCAT (2000)
3. White, S.A.: Business Process Modeling Notation. BPMN 1.0, Business Process Modeling Initiative (2004)
4. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer (2005)

5. Weiss, D.M., Lai, C.T.R.: Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
6. European ESAPS Consortium: ITEA-ESAPS Full Project Proposal (1999)
7. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. 1st edn. Addison Wesley, Reading, Massachusetts, USA (2000)
8. Basset, P.G.: Framing Software Reuse: Lessons learned from the Real World. Yourdon Press (1997)
9. Minsky, M.: A Framework for Representing Knowledge. Mcgraw-Hill, New York (1975)
10. Gacek, C., Anastasopoulos, M.: Implementing Product Line Variabilities. SIGSOFT Softw. Eng. Notes **26**(3) (2001) 109–117
11. Flanagan, D.: Java in a Nutshell. 3rd edn. O'Reilly (1999)
12. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures. Addison-Wesley Professional (2005)
13. Gomaa, H., Webber, D.: Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model. In: Proceedings of the 37th Annual Hawaii International Conference on System Sciences. HICSS'04, IEEE Computer Society Press (2004) 1–10
14. van Gurp, J., Bosch, J., Svahnberg, M.: On the Notion of Variability in Software Product Lines. In: Proceeedings of WICSA 2001. (2001)
15. Bachmann, F., Bass, L.: Managing Variability in Software Architectures. In: SSR '01: Proceedings of the 2001 symposium on Software reusability, New York, NY, USA, ACM Press (2001) 126–132
16. Svahnberg, M., Bosch, J.: Issues Concerning Variability in Software Product Lines. Volume June of 146. Lecture Notes in Computer Science (2003)
17. Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley, Harlow, England et al. (2000)
18. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley (2001)
19. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. Addison Wesley Longman, Harlow, England et al. (1997)
20. Schnieders, A., Puhlmann, F.: Activity Diagram Inheritance. In: Proceedings of the 8th International Conference on Business Information Systems BIS, Poznan, Poland (2005)
21. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison Wesley (1995)
22. Fritsch, C., Lehn, A., Rashidi, R., Strohm, T.: Variability Implementation Mechanisms A Catalog (Internal Paper). Technical report, Robert Bosch GmbH (2002)
23. Speck, A., Clauß, M., Franczyk, B.: Concerns of Variability in 'Bottom-Up' Product-Lines. In: Proceedings of Second Workshop on Aspect-Oriented Software Development, Bonn, Universitt Bonn (2002) 19 – 24
24. PESOA Consortium: PESOA Homepage. (`http://www.pesoa.de`)
25. Giese, C., Buhl, W.: Software Generators (in German). PESOA-Report No. 04/2004. Technical report, Delta Software Technology GmbH (February 2004)
26. Philippow, I., Riebisch, M., Boellert, K.: The Hyper/UML Approach for Feature Based Software Design. In: Proceedings of the 4th AOSD Modeling With UML Workshop. (2003)
27. McNeile, A.T., Simons, N.: State Machines as Mixins. Journal of Object Technology **2**(6) (2003) 85–101

28. Prehofer, C.: Plug-and-Play Composition of Features and Feature Interactions with Statechart Diagrams. Software and System Modeling (3) (2004)
29. Franczyk, B., Riebisch, M.: Extending the UML to Model System Families. In: Proceedings of the IDPT 2000. (2000)
30. Ziadi, T., Hélouët, L., Jézéquel, J.M.: Towards a UML Profile for Software Product Lines. In: PFE. (2003)
31. Jarzabek, S., Zhang, H.: XML-Based Method and Tool for Handling Variant Requirements in Domain Models. In: Proceedings of the 5th IEEE International Symposium on Requirements Engineering, IEEE Computer Society (2001)
32. et al., F.C.: Business Process Execution Language for Web Services version 1.1 (2003)
33. Karastoyanova, D., Leymann, F., Buchmann, A.P.: An Approach to Parameterizing Web Service Flows. In: ICSOC. (2005) 533–538
34. Keller, G., Nüttgens, M., Scheer, A.: Semantic Process Modeling Based on Event-driven Process Chains (EPC) (in German). Veröffentlichungen des Instituts für Wirtschaftsinformatik, University of Saarland, Saarbrücken **Heft 89 (in German)** (1992)
35. Rosemann, M., van der Aalst, W.: A Configurable Reference Modelling Language. Technical Report QUT Technical report, FIT-TR-2003-05, Queensland University of Technology, Brisbane (2003)
36. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: GPCE 2005, Springer-Verlag GmbH (2005)

```
010  public class CalculateSum
020  {
030   public static final boolean PERSONALIZED_SHOPPING_CARD = [PSC-VP1];
040
050   public static void calculateSumSubprocess()
060   {
070         calculateSum();
080
090         if(PERSONALIZED_SHOPPING_CARD)
100         {
110               calculateDiscount();
120         }
130   }
140
150   private static void calculateSum()
160   {
170         …
180   }
190
200   private static void calculateDiscount()
210   {
220         int parameterValue = readParameterValueFromFile();
230
240         switch(parameterValue)
250         {
260               case 3:
270               {
280                     recalculateSumWithDiscount3();
290                     break;
300               }
310               case 5:
320               {
330                     recalculateSumWithDiscount5();
340                     break;
350               }
360         }
370   }
380
390   private static int readParameterValueFromFile()
400   {
410         …
420   }
430
440   private static void recalculateSumWithDiscount3()
450   {
460         …
470   }
480
490   private static void recalculateSumWithDiscount5()
500   {
510         …
520   }
530  }
```

**Figure 15.** Example for Variability Implementation with Java and HyperSenses