Modelling Architectural Variability for Software Product Lines

Thomas Weiler
Research Group Software Construction, RWTH Aachen, Germany
thomas.weiler@cs.rwth-aachen.de

Abstract

In this paper requirements for a concept to model software product line architectures are presented. Furthermore a process for SPL architecture modelling is described which incorporates the concept of the model driven architecture (MDA) into SPL architecture modelling. Besides a metamodel for SPL architecture modelling elements is shown, which – combined with the process for SPL architecture modelling - fulfils the requirements deployed in the first part.

Modelling variability and traceability of requirements within a software architecture thereby possesses the main focus. Therefore a detailed breakdown of different kinds of variability found in product line based software architectures is given. The presentation concludes with an small excerpt from a case-study within the context of an e-shop, which should clarify the application of the elements of the metamodel presented before.

1. Introduction

Software Product Lines (SPLs) are an advancement in software reuse. In the scope of SPLs reuse however refers to all documents that evolve during the development of (similar) products. Examples for these documents are requirements, architecture models or database designs.

SPL development is divided into two main parts, which execute interactively. Within the *domain engineering* the common and variable parts of products, which belong to an *application domain*, are analysed and described. The resulting documents of this process form the basis of the product line, the so-called *Product Line Platform (PLP)*. During the *application engineering* concrete *products* are then derived from this PLP. Thereby the terms *application* and *product* will be used synonymous below.

By maximising the reuse of documents in the product line-based software development, time-to-market as well as development costs can be significantly reduced [1]. Furthermore a correct applied product line-based approach encourages the quality of the end products by careful development and intensive tests of the common parts of the SPL.

2. Present approaches

Most approaches in the scope of SPLs are focusing on the requirements engineering. They primarily consider the delimitation of the application domain during the process of *scoping* as well as the acquisition and modelling of requirements for SPLs.

Thereby it is identified to be crucial, to explicitly model the variability of requirements for products of a SPL. Furthermore a dedicated mechanism is needed, which allows the product developer to resolve the modelled variability for a concrete product in a way desired by the developer of the PLP.

Within all these approaches it is often neglected that product line-based software development can only lead to full success if it is recognized as an integrated concept, which involves all phases of the software engineering process. In the following this article concentrates on architecture modelling for SPLs.

3. SPL architecture modelling

Architecture modelling for SPLs partially demands similar requirements as architecture modelling for *conventional* systems. But many of these requirements need a more intensive attention in the scope of SPLs, because the PLP architecture often forms the basis for a huge set of derived product architectures. This simultaneously is the risk and the chance of SPLs.

In the following requirements for a SPL architecture modelling concept are presented which are determined during the case study presented in section 8 and are additionally the result of a comparison of existing approaches in the context of SPLs, see also section 9. Thereafter a SPL architecture modelling process and a metamodel for SPL architecture modelling elements will be presented which fulfil the specified requirements.

Entities and relations: First of all – as with every other architecture modelling language – there must be a possibility to model the central building blocks of a system – the entities – and their connections, the relations. Thereby the entities describe central units of the system to be modelled and the relations describe structural and behavioural connections of this units like e.g. hierarchical or uses relations.

Separation of concern: Architecture modelling for SPLs must provide the possibility to concentrate on specific aspects of a system [10]. This concept known as *separation of concern* is divided into two dimensions: Along the *horizontal dimension* it is possible to designate the focus on a part of interest (*clipping*). The *vertical dimension* allows to magnify a given fixed cutout step by step in order to get a more and more exact image of the cutout in question.

A combination of both dimensions is the so-called *zooming*, in which an aspect is magnified step by step whereby the observed cutout is simultaneously scaled down and vice versa. This may be seen analogous to a photographic lens with zoom-function where a longer focal length (higher magnification) results in a smaller angle.

Traceability: Traceability of requirements down to the architecture and finally to the source code (and back) is a vital task to ensure the comprehensibility and maintainability of a software system. In the scope of SPLs the claim for traceability is so much important because resolving the variability of the requirements has direct impact on the design and therefore the source code of the SPL. Only if the traceability of requirements down to the design and furthermore the source code is guaranteed, one can fully benefit from the possibilities of reuse and therefore of cost-saving.

Evolution: Similar to conventional software products a SPL isn't resistant against changes during its life cycle. By and by changing requirements lead to changed architectures and products. Therefore a mechanism is needed to track these changes over time. In the context of SPLs this not only means versioning but also to decide when and how to migrate already derived products when changing the PLP.

Technical platform independence: To maximise the benefit of reusing components, the design of a system and components respectively should be independent of the implementation technique used as long as possible along the levels of abstraction. Thereby the term *component* is not meant to denote a component known from e.g. CORBA or EJB but a higher building block used in architecture modelling. This will be discussed in more detail in section 6.

The request for technical platform independence complies with the *Model Driven Architecture (MDA)* approach conceived by the OMG [4]. In the scope of architecture modelling for SPLs, this technical platform independence refers to the development of the PLP architecture as well as the architectures of therefrom-derived products.

It should be mentioned that the term *platform* is used in the scope of SPL engineering as well as in the *MDA* approach. So one should not mix up the two meanings of the term *platform*. While in the context of SPLs this term describes all documents on which the product line is based, in the context of the MDA it refers to the *technical platform* used. So if not explicitly mentioned context should clarify which meaning was meant by. The relationship between SPLs and the MDA will be discussed in more detail in sections 4 and 5.

Variability: Modelling different variability within a SPL is vitally important for the requirements engineering as well as for designing the architecture. Combined with the *traceability* arises the possibility to resolve variability at the level of requirements during product configuration and to implement it through the design level down to the implementation level, see also section 4. Therefore a concept for SPL architecture modelling needs to provide the possibility to distinguish between common and variable parts of the products derived from a PLP.

Decision support: In order to resolve variability offered in the PLP architecture in a way intended by the platform developer a mechanism is needed, which helps the product developer to make the needed decisions. Therefore each variability modelled in the PLP architecture must be furnished with an annotation – normally formulated in natural language – which provides the product developer with the needed information to resolve given variability.

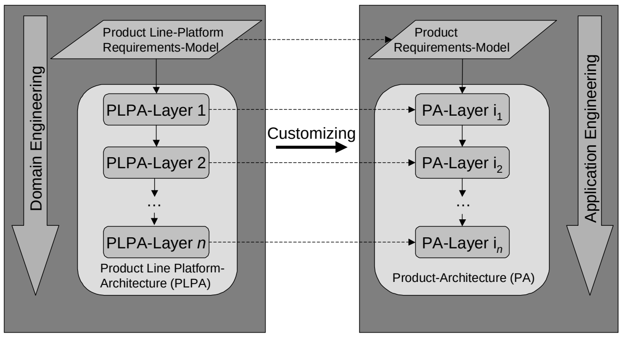
Dependencies: By modelling the variability within a SPL it must be taken into account, that there might be dependencies between components of the system. This can mean that for example the existence of one component requires the existence of another component. Therefore a concept for SPL architecture modelling needs to support an appropriate type of relationship.

Having described the requirements for SPL architecture modelling in the next section a process will be presented, which illustrates the necessary steps and the dependencies by modelling SPL architectures.

4. SPL architecture modelling process

This section presents a process for SPL architecture modelling. As already mentioned in section 1 SPL architecture modelling is organized in the two areas *domain engineering* and *application engineering*. In Figure 1 the part of architecture modelling gets more improved.

Within the *domain engineering* initially the requirements for the entire PLP are collected together with the identified variability and afterwards compiled into a *requirements model* for the PLP, which among other things contains e.g. a *feature graph* [2]. This requirements model forms the basis for the top-level layer of the PLP architecture. Starting from this still abstract architecture layer the PLP architecture gets more and more improved in further architecture layers. This procedure is according to the *Model Driven Architecture (MDA)* approach introduced by the OMG [4], see also section 5.



Product Line-Platform Requirements-Model
Product Requirements-Model
Domain Engineering
Application Engineering
PLPA-Layer 1
PLPA-Layer 2
PLPA-Layer n
Product Line Platform-Architecture (PLPA)
Customizing
PA-Layer i1
PA-Layer i2
PA-Layer in
Product-Architecture (PA)

Figure 1. SPL architecture modelling process

In the last step within the domain engineering the that way specified generic architecture gets realized as far as possible. Thereby – according to the differentiation in common and variable components – both finished and incomplete components are placed in the PLP, see also section 6.

At the beginning of the *application engineering* firstly the requirements for a concrete product are determined on base of the requirements for the PLP. Afterwards – similar to the domain engineering – a first coarse architecture layer for the product is developed, which is based on the layer of the same abstraction level as in the PLP architecture. In the following this top-level architecture becomes more and more improved analogue to the layers of the PLP architecture.

Thereby the variability included in the PLP architecture is resolved conform to the previously identified product requirements. In the last step the executable system is implemented based on this product architecture.

5. MDA and SPL architectures

To fulfil the requirement of technical platform independence - see section 3 - the *Model Driven Architecture (MDA)* approach of the OMG [4] can be incorporated into a model for SPL architecture modelling. Figure 2 shows an approach to integrate the MDA in a concept for modelling SPLs.

Thereby the *core model* known from the MDA is specialized to a *domain specific core model*, which offers modelling elements adapted on a given domain. These modelling elements are used to define a *platform independent PLP model* conforming to the MDA, based on the analysed requirements for the PLP. The platform independent PLP model consists of several *abstraction layers*, which give from top to bottom a more and more complete view of the modelled system. It is then - according to the MDA - mapped to a *platform specific PLP model*, which also consists of several abstraction layers.

During the *application engineering* initially the product requirements are determined based on the requirements of the PLP and then implemented by a *platform independent product model* pursuant to the

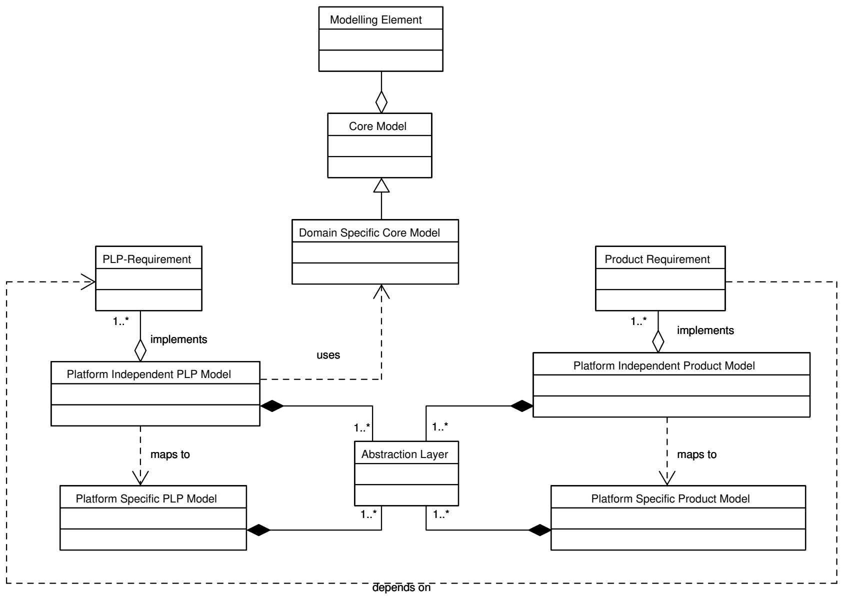


Figure 2. MDA and SPLs

MDA. This consists – analogue to the platform independent PLP model – of several abstraction layers and is mapped to a *platform specific product model,* which in turn consists of several abstraction layers.

6. Feature components

The central building blocks for modelling the PLP and application architectures in the approach presented here are *feature components*. A feature component can be seen as a self-contained unit, which represents a specific characteristic of the system to be modelled. They are an adaptation of the *feature* concept introduced by the *Feature Oriented Domain Analysis (FODA)* to the level of architecture modelling for SPLs [1].

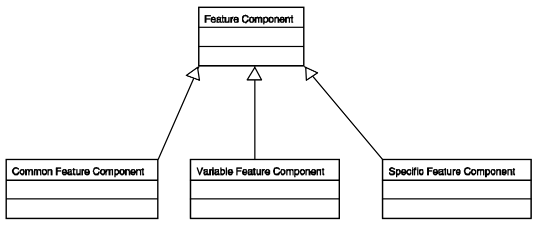


Figure 3. Feature Components

It must be mentioned that the feature components at the level of architecture modelling aren't necessarily identical to the features according to FODA, which are identified at the level of the requirements analysis [2]. For example it might be possible that a set of features identified in the requirements analysis together build a feature component at the level of architecture modelling. It might also be possible, that a feature is implemented by a set of feature components likewise *aspects* in the *Aspect Oriented Development* [5]. Furthermore feature components need – contrary to their name – not to be realised at the implementation level as components provided by for example CORBA or EJB. As shown in Figure 3 feature components can be divided into three different types.

Common feature components are used in a PLP architecture and describe feature components, which can occur in every application based on this architecture. Common feature components occur in derived application architectures without modification.

Variable feature components are feature components, which can occur in every derived application architecture only by resolving the offered variability of type *incomplete specification*. This type will be described in more detail in section 7.1.

The last type of feature components is represented by *specific feature components*. They are special building blocks needed to construct a specific application architecture derived from a PLP architecture. At this it must be taken into account, that in the course of the evolution of a SPL an initially product-specific feature component at a later date can be incorporated into the PLP and thereby become a variable or even a common feature component of the PLP, see section 3.

7. Metamodel

After this preparatory work in this section a metamodel for SPL architecture modelling elements will be given which – in conjunction with the SPL architecture modelling process presented in sections 4 and 5 – fulfils the requirements described at the beginning. In section 8 an example will illustrate the elements presented in the metamodel shown in Figure 4.

The central modelling element is the *feature component* mentioned in section 6. Thereby each feature component memorises the requirements covered by it. In doing so *traceability* of requirements down to the architecture level is supported as asked for in section 3.

Feature components can participate in *relations* with the aid of *relation ends* as known from the UML [3]. Thereby a relation can be a *dependency* – see also section 3 – or a *hierarchy* relation.

Among a *dependency*-relation two different kinds of dependencies between feature components can be distinguished:

- Prohibited
- Required

A dependency of type *prohibited* is an undirected relationship between two feature components. In a *prohibited*-Relationship the existence of one feature component forbids the existence of the other feature component in a derived product architecture.

A dependency of type *required* is a directed relationship between two feature components. It is used if the existence of one feature component of the PLP architecture depends on the existence of another feature component of the PLP architecture within a derived product architecture.

A *hierarchy*-relation depicts a conceptual structure between a super- and a – possibly set of – sub-feature component(s). It should be seen more as a *is part of*-relation than a generalisation similar to the connections used in a *feature graph* in FODA [2].

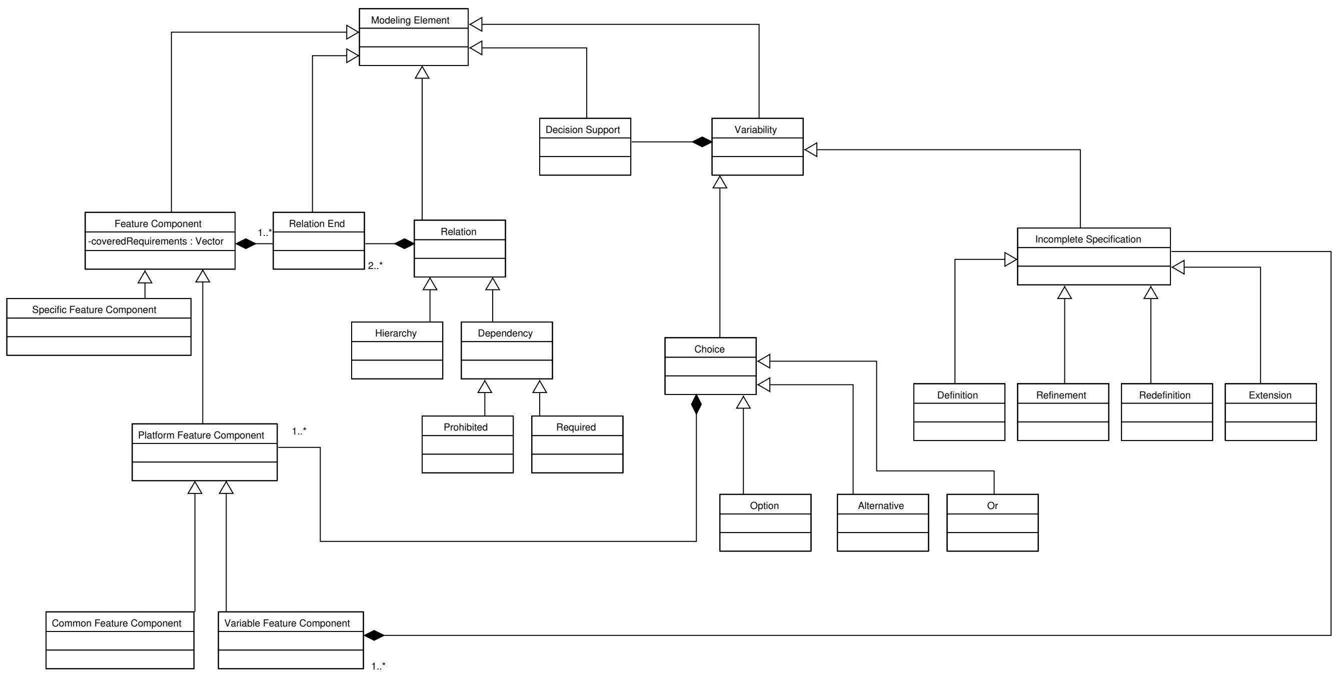


Figure 4. Metamodel for SPL architecture modelling elements

The other major part of the metamodel pertains to the modelling of *variability*. Thereby two types of variability can be distinguished: *incomplete specification* and *choice*.

7.1. Incomplete specification

Variability in the form of an *incomplete specification* is characterised by a missing or incomplete specification of a component. At this four different types can be distinguished:

A *definition* only determines the skeleton of a feature component likewise an interface. The detailed specification is done during the *application engineering*.

A *refinement* defines the behaviour and data of a feature component in an abstract way likewise a template- or hook-feature component. The exact design will be defined product-specific.

At the *redefinition* a specification for the feature component exists already but it can be renewed product-specific. This can serve for the definition of a preset specification of a feature component, which can be product-specific redesigned.

Similar to the *redefinition* the *extension* also defines a (standard) specification of a feature component. However this specification can be product-specific extended by functions or data.

Beyond these four types of incomplete specification *redefinition* and *extension* are *optional* variability because in these cases a sufficient complete specification of the feature component in question is given. On the other hand variability of type *definition* or *refinement* must always be resolved.

7.2. Choice

The second type of variability between members of a SPL concerns the *choice* from a set of offered feature components from the PLP. It can be distinguished in the following three types:

- Option
- Alternative
- Or

In case of an *option* the product developer has to decide, if he takes over an optional feature component from the PLP to the product architecture. In case of an *alternative* exactly one feature component must be chosen from a set of offered feature components.

An *or*-choice describes a set of feature components from which one ore more feature components must be chosen. Table 1 shows the different types by illustrating the used cardinalities of the choice and selection sets. It should be mentioned that these three types could also be combined to obtain a broader variety of possible sets to choose from.

Table 1. Choice

	Cardinality of choice	Cardinality of selected set
Option	0..1	1
Alternative	1	*
Or	1..*	*

When resolving variability during the *application engineering*, incomplete specifications must be completed that means defined, refined, redefined or extended. Furthermore the product developer has to come to a decision about the feature components to choose from sets of offered feature components in variability of type choice.

Regarding all types of variability a *decision support* is provided which supports the product developer resolving given variability, see section 3.

8. Example

In the following a small excerpt from a first case-study is presented to illustrate the application of the metamodel elements. This case study models a SPL in the context of an Internet e-shop.

In Figure 5 a feature graph modelling the *order* subsystem of an e-shop product line is shown. Thereby an extended notation compared to FODA is used [2].

The order system consists of an optional feature *payment* denoted by the circle above the feature element. The feature graph defines different types of payment methods among which the product developer can chose one or more. Within this *or*-choice – see section 7.2 – the feature *other payment method* is a placeholder for further payment methods which can be defined product specific.

On the right hand of the feature graph a feature *order confirmation*, which denotes the kind of order confirmation for the seller, is described, where the product developer must decide, which one of the alternatives offered he chooses, see also section 7.2. Amongst the three offered alternatives the feature *fax* needs to be redefined in a derived application, see section 7.1.

The two remaining optional features are the possibility to distinguish a *delivery address* from a billing address and to make use of a *gift service*. Thereby the *gift service* depends on the feature *delivery address* because one rarely wants to send one's gift together with an invoice. This is shown by the use of a *requires* relationship between this two features.

In the feature graph shown every variability is numbered, whereby the numbering scheme should be read from top to bottom. For example the variability of type *definition* at the feature *other payment method* has number 1.1b.1 because it is under the or-choice number 1.1, which in turn is under the optional feature *payment*, which has number 1.

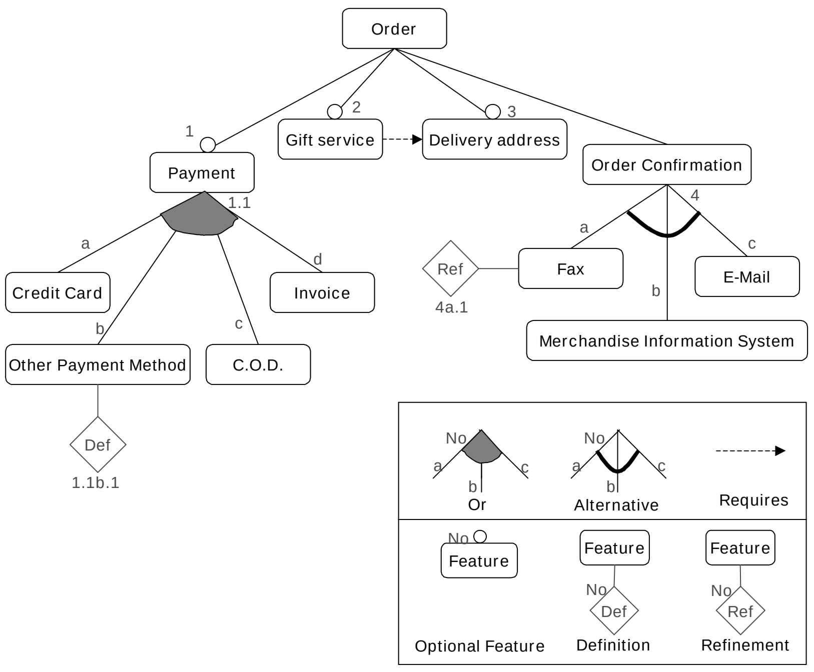


Figure 5. Feature graph e-shop

By using this numbering scheme the product developer can move along a *decision tree* build up from this hierarchical variability numbers. Together with a *decision support* for every variability modelled, that way the product developer can easily resolve the variability offered by the PLP.
After this description of an feature graph for the *order* part of the e-shop the associated PLP architecture will be presented in part. It is constructed as a three-layer architecture.

The PLP architecture is made up of a *presentation layer,* which visualises the outcomes of the subjacent *business logic layer* and serves in addition as the communication interface from the end user to the e-shop system, normally by means of a web browser.

The *business logic layer* contains the functional components of the e-shop, e.g. order handling or customer management. In the following this layer will be described in more detail.

The lower most layer is the *database layer*, which provides the business logic layer with the functionality needed to manage the dates with the help of a database system.

It should be mentioned that the layers described here aren't identical to the PLP architecture layers mentioned in sections 4 and 5. Here the three layers describe a logical segmentation of the system to be modelled (a tier-architecture) whereas in the second case the layers describe the hierarchy of abstraction of the modelled PLP architecture.

The variability described in the feature graph in Figure 5 is brought down to the PLP architecture of the e-shop. Figure 6 presents a part of the business logic layer, which amongst other things consists of the feature components *order_system, data_access_support, customer_management, application_control,* and *catalog_management.*

It is visible that the feature component *order_system* is influenced by two types of variability presented in the feature graph in Figure 5. Furthermore the feature component *catalog_management* has a variability annotated, which was modelled in another here not shown part of the feature graph.

The feature component *data_access_support* in the above figure shall depict a feature component, which has no direct conjunction with features from the feature graph but is a feature component needed for technical realisation. It should be mentioned that it is possible, that certain variability arises not until architecture level. Thus it is imaginable, that a feature component can be realised in many different ways – for example a DBMS can be realised relational or object oriented.

The two other feature components in Figure 6 will not deepened and are only shown for reasons of completeness. In the following the feature component *order_system* will be observed in more detail.

Figure 7 shows a detailed view of the feature component *order_system* mentioned before. Here the abstraction level allows using a well-known modelling language – here the UML – in order to describe the specific characteristics of this feature component. As can be seen in Figure 7 the different types of variability modelled in conjunction with the features *payment* and *order confirmation* in the feature graph of Figure 5 can be regained in the feature component *order_system*.

The optional feature *payment* is mapped to the now optional class *PaymentMethod* depicted by the circle with annotation *Opt* and number 1. Similar the alternative number 4 and the or-choice number 1.1 are represented in this feature component. Three additional classes are shown, which describe an order based on a (virtual) shopping cart. These two classes come from another feature not modelled in the feature graph shown in Figure 5.

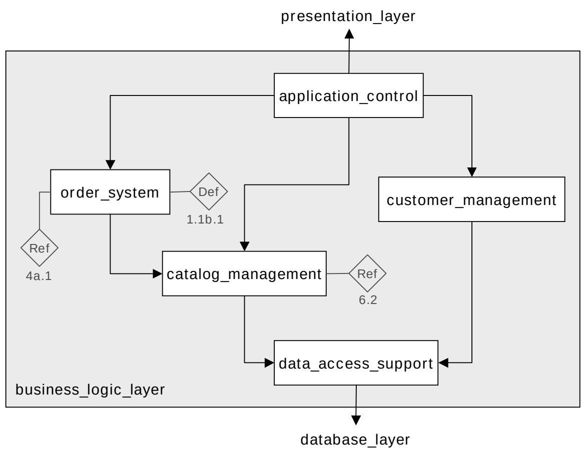


Figure 6. Business logic layer

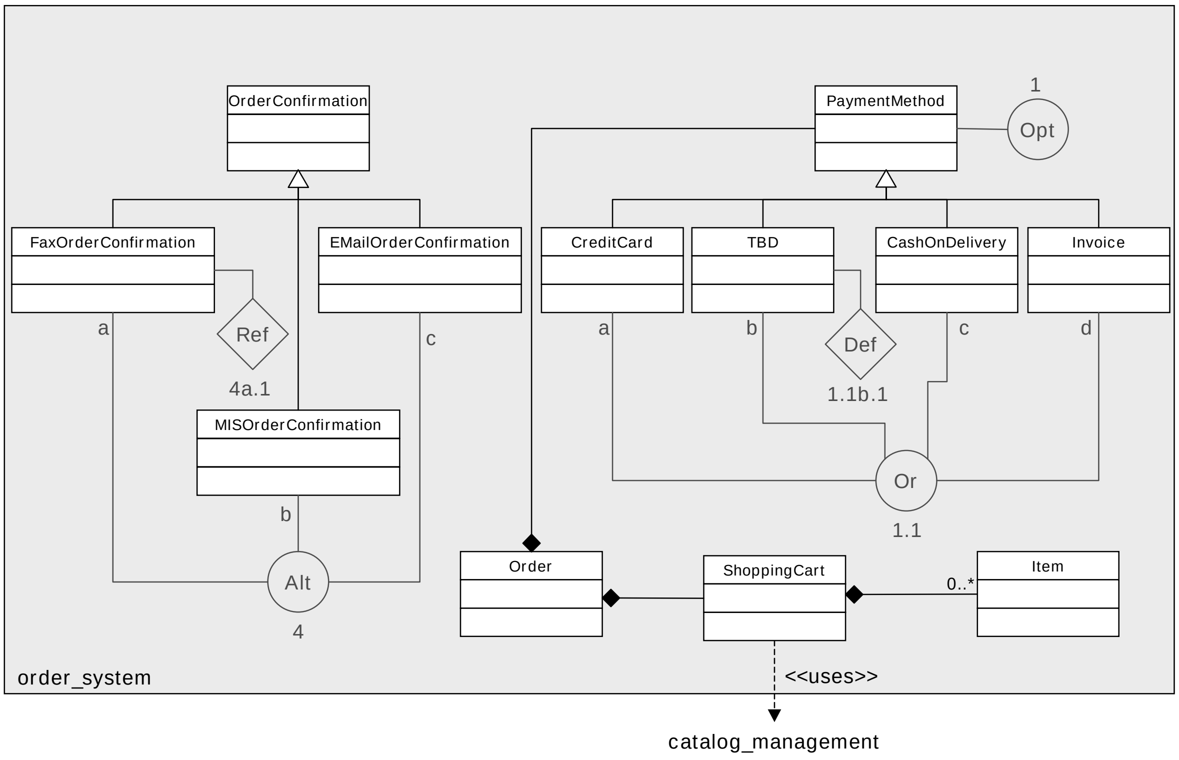


Figure 7. Order system

It should be pointed out that the types of variability shown in the feature graph not only have impact on the *business logic layer* and therefore the feature component *order system* but also on the other layers *presentation layer* and *database layer* and their corresponding feature components. For example the or-choice number 1.1 between the different types of payment methods must also be modelled (and implemented) at the *presentation layer*, so that e.g. the end user can choose his preferred payment method. As can be seen in this example, the mapping of features from the feature graph doesn't need to match one-to-one with the feature components modelled at the architecture level, as already mentioned in section 6.

The next step is to bring the modelled variability down to the source code. This can be achieved by annotating the source code with appropriate tags to depict the different types of variability. Because this actual is work in progress it will not deepened here.

9. Related work

As stated in section 2 most of the existing approaches concerning SPLs are focusing on the requirements engineering. Nevertheless some approaches exist which try to concentrate more on the downstream phases of the development process like the design, whereby some of them had certain influence on the approach presented in this article. As also stated by Muthig et.al. in [8] existing approaches often seem to be more pragmatic solutions resulting from practical modelling experiences in a particular domain or environment whose results are not universally transferable.

In [6] Flege describes an approach for using the UML [3] for system family architecture description. Thereby he focuses solely on construction-time variability, because only this type of variability results in different products and is therefore essential for developing SPLs. Presence of variability at later stages like e.g. at binding or runtime doesn't require special attention in the context of SPLs because they only affect one single product, see also [8] and [9].

The drawback of Flege's approach is the lack of elements in the UML for explicit modelling of architectural variability. Flege uses UML's stereotypes to depict variable architectural elements. Thereby he only models optional elements by neglecting e.g. alternatives among modelling elements. In Flege's approach alternatives should be modelled at the level of the decision model. At the design level this leads to optional elements (the single alternatives) which are no more distinguishable from other, real optional elements. Therefore the approach presented in this article explicitly distinguishes the different types of variability presented in section 7 at the design level to allow traceability from the requirements down to the design and the source code.

Furthermore Flege focuses exclusively on variability with a complete set of specified variants by discarding variability of type incomplete specification that might be used by product developers in an unanticipated way. As per Flege the reason for this is that unspecified variability has no impact during the instantiation of a reference architecture. In the approach presented in this paper variability of type incomplete specification is explicitly included. At first different specifications of elements among products of a SPL – resulting in incomplete specification in the PLP architecture – are a distinguishable characteristic of these products and therefore represent one type of variability within a SPL. Furthermore only by explicitly modelling variability of type incomplete specification – including the corresponding decision support – one can help the product developers to use the offered variability only the way intended by the PLP developers.

In [7] Batory et.al. refer to the need for higher-level modelling elements when modelling SPL architectures. Therefore they use features at the design level instead of e.g. modules. These features are then step-wise refined during the design resulting in a more and more precise architecture description. In their approach Batory et.al. concentrate more on the transition from the design to the implementation by introducing templates for JAVA. The *feature components* presented in section 6 also try to offer higher-level architecture modelling elements but are – contrary to Batory et. al. – clearly differentiated from the features of FODA [2] used during the requirements analysis.

10. Conclusion and future work

In this paper requirements for a concept to model SPL architectures were presented. Furthermore a SPL architecture modelling process was described which incorporates the concept of the *model driven architecture* into SPL architecture modelling. Besides a metamodel for SPL architecture modelling elements was shown, which – together with the described SPL architecture modelling process - fulfils the requirements deployed in the first part.

A first practical application in the context of a case-study from which parts were shown in the example illustrated in section 8 has shown the load capacity of the presented concepts for a medium sized application. Within this case-study a domain for e-shops was analysed and based on a requirements model including a feature graph for this domain a PLP architecture using the modelling elements offered by the presented metamodel was developed.

For the time being two products were derived from this PLP to show the load capacity of the given concept. Thereby it turned out that – although the concept was useful – a meaningful and broader application can only be achieved if the concepts are supported by tools. Otherwise the PLP and product developers can hardly manage the given complexity.

This leads to another aspect, which requires more work to be done: The transitions from requirements engineering to architecture design and from architecture design to the level of implementation must be supported in a concept for modelling SPL architectures. Otherwise the lack of systematics makes the stability and durability of a SPL solely depending on the intelligence and creativity of the developers involved.

11. References

[1] Donohoe P. (editor), *Software Product Lines: Experience and Research Directions*, Kluwer International Series, 2000.

[2] Kang, et. al., *Feature Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report SEI-CMU, Pittsburgh, 2000.

[3] OMG, *Unified Modeling Language Specification, Version. 1.4*, Technical Report, OMG, 2001.

[4] Soley R., OMG, *Model Driven Architecture*, White Paper, OMG, 2000.

[5] AOSD Steering Committee, *Aspect-Oriented Software Development*, http://aosd.net

[6] Flege O., *System Family Architecture Description Using the UML*, IESE-Report No. 092.00/E, 2000

[7] Batory, Johnson, MacDonald, and von Heeder, *Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study*, ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 11, Nr. 2, pp. 191-214, 2002

[8] Muthig and Atkinson, *Model-Driven Product Line Architectures*, SPLC 2002, LNCS 2379, pp. 110-129, 2002

[9] Thiel S. and Hein A., *Systematic Integration of Variability into Product Line Architecture Design*, SPLC 2002, LNCS 2379, pp. 130-153, 2002

[10] van Zyl, *Product Line Architecture and the Separation of Concerns*, SPLC 2002, LNCS 2379, pp. 90-109, 2002